SYBEX

THE COMMODORE 64™/ VIC-20™ BASIC HANDBOOK

DOUGLAS HERGERT

THE COMMODORE 64/VIC-20 BASIC HANDBOOK

THE COMMODORE 64™/VIC-20™ BASIC HANDBOOK

DOUGLAS HERGERT

SYBEX®

Berkeley • Paris • Düsseldorf

Book design by Ingrid Owen
Cover art by Jean-Francois Pénichoux

Library of Congress Card Number: 83-50718
ISBN 0-89588-116-0
First Edition 1983
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ACKNOWLEDGEMENTS

Once again I want to thank James Compton for a skilled and genial editing job. My thanks also go to Guy Orcutt for his painstaking work on the screen photography; and to the following members of the Sybex editorial and production staff: Valerie Robbins and Donna Scanlon, word processing; Valerie Brewster, typesetting; Ian Hughes, Janis Lund, and Laura Bennett, proofreading; and Ingrid Owen, design and artwork.

CONTENTS

INTRODUCTION

This book is designed to help you master the features of BASIC on your computer. The entire Commodore 64™ and VIC-20™ BASIC vocabulary is included, along with the disk and tape commands. The entries are presented, not in formal dictionary style but in a practical format that should be easy to understand and to use. In general, the entries are organized into the following sections:

— a description of a given BASIC command or function—what it does, and how to use it correctly in a program;

— a sample program and an explanation of how the program works, focusing specifically on the BASIC word being illustrated;

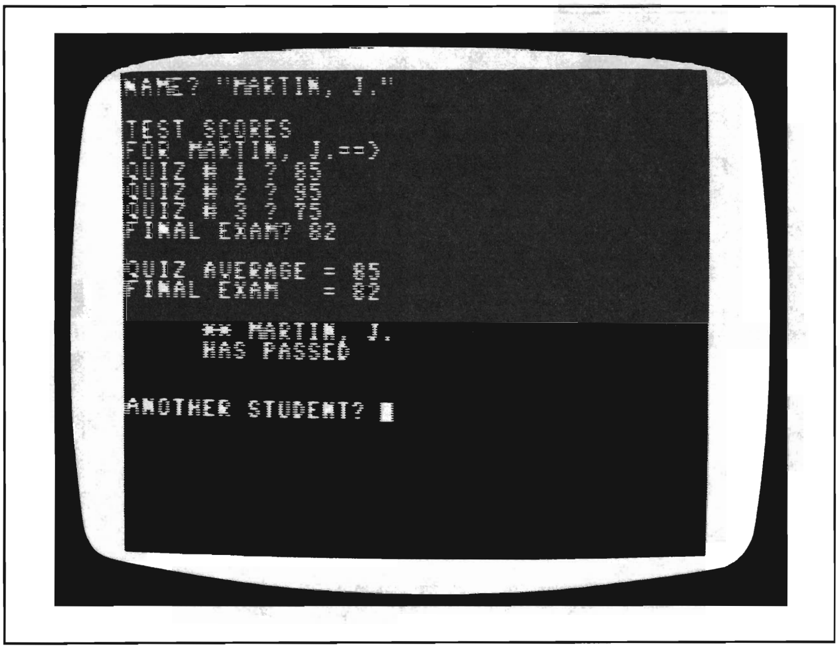
— a screen of output from the sample program;

— "Notes and Comments," an assortment of interesting sidelights and items of practical information—for example, extended uses to experiment with; errors to watch out for; other BASIC words to study in connection with the word at hand.

This format sometimes varies to fit the needs of individual entries. In all cases, the goal of this format is to incorporate the BASIC command words into your *active* programming vocabulary, so that you can begin using the commands in your own programs.

You will probably find yourself paying particular attention to the sample programs in this book; the features of any computer programming language are often easier to learn in the context of real examples than in the abstract of descriptive prose. Consequently, you should enter the programs into your computer and run them, in order to benefit fully from their educational value. Each program is designed to illustrate the characteristics, the subtleties, and sometimes the quirks of a given BASIC command word. These programs are learning exercises and should be used as such; their main goal is to serve as a medium of instruction. All the same, some of them may prove to be useful or amusing in their own right. For example, among the programs of this book, you will find:

- a program that creates bar graphs from numerical data that you enter at the keyboard (*see* DIM);
- programs that display various kinds of graphics on the video screen (*see* FN, STEP);
- a program that will get you started in writing any computerized card game (*see* RND);
- a guessing-game program (*see* IF);
- programs illustrating "menus" and "user-friendly" input (*see* GOSUB and GET);
- a program that helps you balance your checkbook (*see* GOTO);

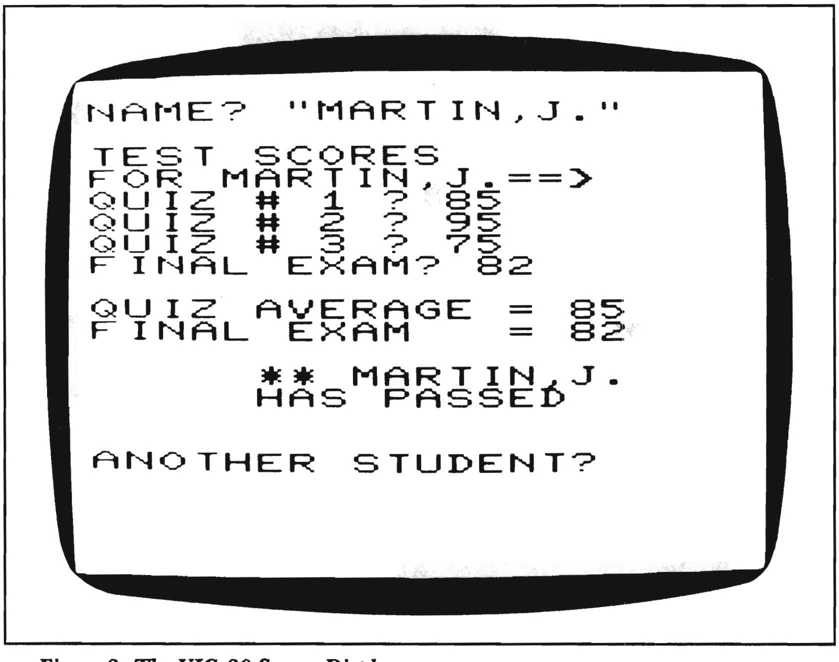


```
NAME? "MARTIN, J."

TEST SCORES
FOR MARTIN, J.==>
QUIZ # 1 ? 85
QUIZ # 2 ? 95
QUIZ # 3 ? 75
FINAL EXAM? 82

QUIZ AVERAGE = 85
FINAL EXAM   = 82
      ** MARTIN, J.
      HAS PASSED

ANOTHER STUDENT? █
```

Figure 1: The Commodore 64 Screen Display

- — programs that help you explore the organization of your computer's memory (*see* PEEK and POKE);
- — a program that converts numeric values into dollar-and-cent display strings (*see* STR$).

For the sake of legibility, SYBEX has produced the output display screens shown in this book by running the programs on a VIC-20 computer, but the programs will work equally well on a Commodore 64. BASIC is essentially identical on the two machines. Programs written for one machine can generally be run on the other (as long as no reference is made to specific memory addresses). The most obvious difference between the computers is the width of the screen display: the VIC-20 has a 22-column screen, whereas the Commodore 64 has a 40-column screen. (The photographs in Figures 1 and 2 illustrate the difference in the screen displays. Figure 1 shows a display produced by the Commodore 64. If you run the programs in this book on that machine, they will produce similar displays. Figure 2 shows the same display, as produced by the VIC-20. If you run the programs on the VIC-20, the display will be as shown here and throughout the book.) Because of this difference, if you are running these



```
NAME? "MARTIN,J."

TEST SCORES
FOR MARTIN,J.==>
QUIZ # 1 ? 85
QUIZ # 2 ? 95
QUIZ # 3 ? 75
FINAL EXAM? 82

QUIZ AVERAGE = 85
FINAL EXAM   = 82

        ** MARTIN,J.
        HAS PASSED

ANOTHER STUDENT?
```

Figure 2: The VIC-20 Screen Display

programs on a Commodore 64 computer, you may want to make minor adjustments to take advantage of the wider screen. Often this is simply a matter of substituting a new value for the argument of a TAB function. In addition, specific system values have different memory locations in the two computers. In most BASIC programs this difference is irrelevant; BASIC, after all, is designed, for the most part, to make the innermost organization of the computer system invisible to the programmer. However, in the few contexts in which memory locations are important in this book, the memory differences between the two computers are explained. (*See* GET, PEEK, POKE, USR.)

Finally, the entries in this book include a number of general computer terms you will want to learn as you get involved in programming on your Commodore 64 or VIC-20 computer. A list of these terms appears in Figure 3. Generally, the definitions in this book avoid unnecessary computer jargon; but some terms, such as the ones in this list, are in common enough use that it is to your benefit to learn what they mean as you are mastering the vocabulary of BASIC. These terms are *italicized* the first time they are used in the definition of any other term or command.

Algorithm	Interactive
Argument	Literal Value
Arithmetic Expression	Logical Expression
Array	Machine Code
BASIC	Menu
Byte	Program
Concatenation	Programmer
Cursor	Scientific Notation
Error Message	String
File	Subroutine
Function	User Friendly
Immediate Command	Variable

Figure 3: General Programming Vocabulary Defined in this Book

ABS (function)

The ABS *function* supplies the absolute value of a number. The *argument* of ABS may be a *literal* numeric *value*, a *variable*, or an *arithmetic expression*. ABS returns the unsigned magnitude of the resulting value.

Sample Program

ABS is useful whenever the sign (negative or positive) of a number is irrelevant, as illustrated in the program shown in Figure A.1. This program is a simple exercise in which pairs of random numbers are chosen and compared. For each comparison, the variables R1 and R2 store the two numbers. Line 90 finds and displays the difference between the two numbers:

```
90 PRINT "IS"; ABS(R1-R2);
```

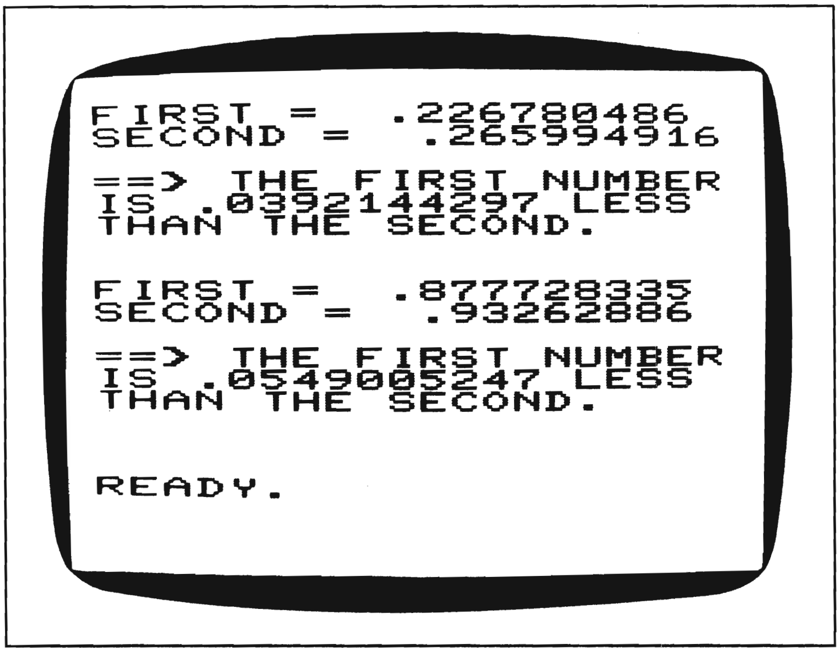
Since the two numbers are chosen randomly (by the RND function), there is no way of knowing which will be larger. Therefore, the expression:

```
R1-R2
```

may result in either a negative or a positive number. But in describing the difference between R1 and R2, the sign is irrelevant, so we use the ABS function to eliminate the sign. Figure A.2 shows a sample output from this program. For each pair of random numbers, the absolute value of their difference is given.

```
 10 PRINT CHR$(147)
 20 FOR I = 1 TO 2
 30   LET R1 = RND(0)
 40   LET R2 = RND(0)
 50   PRINT "FIRST = ";R1
 60   PRINT "SECOND = ";R2
 70   PRINT
 80   PRINT "==> THE FIRST NUMBER"
 90   PRINT "IS"; ABS(R1-R2);
100   IF R1<=R2 THEN PRINT "LESS"
110   IF R1>R2 THEN PRINT "GREATER"
120   PRINT "THAN THE SECOND."
130   PRINT : PRINT
140 NEXT I
150 END
```

Figure A.1: ABS—Sample Program



```
FIRST =  .226780486
SECOND =  .265994916

==> THE FIRST NUMBER
IS .0392144297 LESS
THAN THE SECOND.

FIRST =  .877728335
SECOND =  .93262886

==> THE FIRST NUMBER
IS .0549005247 LESS
THAN THE SECOND.


READY.
```

Figure A.2: ABS—Sample Output

Algorithm (computer vocabulary)

An algorithm is a series of steps designed to accomplish a defined task.

We often begin the process of planning a BASIC program by expressing an algorithm in words, before attempting to write it as a sequence of BASIC instructions. For example, consider the following steps:

1. Increase the value of the variable V by 5.
2. Display the new value of V on the screen.

These two steps can be translated into the following two program lines:

```
10 LET V = V + 5
20 PRINT V
```

Sometimes, however, what seems like a simple algorithm when expressed in words will turn out to be much more complicated when implemented as a BASIC program. Consider, for example, the following algorithm:

1. Read ten words from the keyboard.
2. Alphabetize the words.
3. Display the words in alphabetical order on the screen.

While these three steps may seem simple and straightforward enough expressed in this way, you will find that it requires no fewer than fifteen program statements to carry them out successfully. A program that performs these steps appears in Figure A.3.

In the case of this algorithm, you may find that you'll have to return to the steps as you originally expressed them, and think them through in more detail before you undertake to write the program. The problem is that your original three steps involve larger tasks than BASIC can handle in single statements. For example, think of the second step: "Alphabetize the words." If BASIC had a command ALPHABETIZE in its vocabulary, you might be able to write a statement such as:

```
20 ALPHABETIZE (W$)
```

Unfortunately, no such command exists, so you must think through the detailed instructions that are required to make the computer perform the alphabetization task. You might replace step 2 with the following, more detailed steps:

2a. Compare each word in the list, one at a time, with each of the words below it in the list.

2b. If the two words in any given comparison are found to be out of alphabetical order, then swap their places in the list.

These two steps bring you closer to the actual sequence of BASIC instructions that you must write; even so, the level of detail is not yet precise enough. Eight program lines (50 to 120) are required to perform these two steps. (These lines represent what is called a "sorting" algorithm.) Again, you must examine the language of your algorithm expression and find where you have oversimplified the steps of the process. For example, consider the phrase, "swap their places in the list," in step 2b. Most BASICs lack the command that would allow you to write:

```
80 SWAP (W$(I),W$(J))
```

so instead, you must think through the three steps required to perform the swap:

2b(1) Store the first word in a "holding variable," H$.
2b(2) Store the second word in the place of the first word.
2b(3) Store the value of H$ in the place of the second word.

These three steps are performed in the three program lines numbered 80 to 100.

In summary, the process of determining the steps of an algorithm may often require successive "magnifications" until the level of detail corresponds roughly to steps that can be translated into BASIC commands.

```
 10 DIM W$(10)
 20 FOR I = 1 TO 10
 30   INPUT W$(I)
 40 NEXT I
 50 FOR I = 1 TO 9
 60   FOR J = I + 1 TO 10
 70     IF W$(I) < W$(J) GOTO 110
 80     LET H$ = W$(I)
 90     LET W$(I) = W$(J)
100     LET W$(J) = H$
110   NEXT J
120 NEXT I
130 FOR I = 1 TO 10
140   PRINT W$(I)
150 NEXT I
```

Figure A.3: Algorithm—Sample Program

AND (logical operator)

The logical operator AND can be used to create a compound *logical expression* for an IF decision. The value, true or false, of such a compound expression depends on the values of the elements that are combined by AND. A compound expression in the following form:

```
statement-1 AND statement-2
```

is true if and only if statement-1 and statement-2 are both true. If either statement is false, or if both are false, then the compound expression is also false.

Sample Program

The program shown in Figure A.4 illustrates the use of AND. It is designed for the following hypothetical situation: A classroom teacher is looking at a semester's test scores to see which students have passed and which have failed. The teacher has given three quizzes and one final exam during the semester. The teacher has decided that a student must have an average score of 75 or better for the quizzes and a final exam score of 70 or better to pass the course. Using this program, then, the teacher can type each student's name and test scores at the computer's keyboard, and the computer will make the appropriate calculations to determine whether the student has passed or failed.

Lines 10 to 110 read the information for a given student. Notice in particular the FOR loop at lines 50 to 90, which reads each quiz score and accumulates the total in the *variable* QT. Line 100 then assigns the average of the quiz scores to the variable AVE. Line 110 reads the final exam score, assigning it to the variable F.

Lines 130 to 190 display the student's test information on the screen. Line 170 illustrates AND:

```
170 IF AVE >= 75 AND F >= 70 THEN PRINT "PASSED"
```

The action of this IF statement is to print the word PASSED on the screen, but only if both of the following statements are true:

```
AVE >= 75
F >= 70
```

In other words, the student passes only if the average quiz score (AVE) is greater than or equal to 75, and the final exam score (F) is greater than or equal to 70. Otherwise, if either one of these conditions is not met, or if neither is, then line 170 results in no action, and the program continues on to line 180.

Figures A.5 and A.6 show the scores for two different students. The first student passed the course by satisfying both conditions. The second student had a satisfactory quiz score, but received a score below 70 on the final exam; the compound statement in line 170 is evaluated to false, and the student fails.

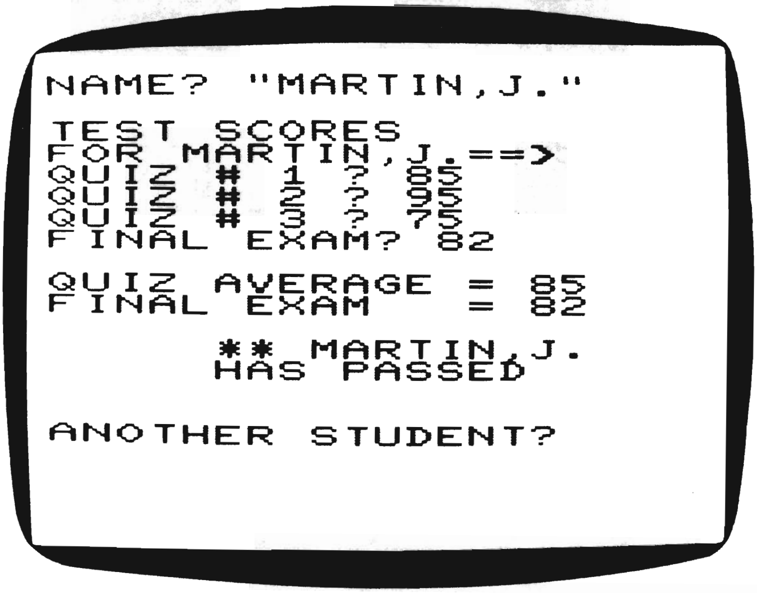
```
  5 PRINT CHR$(147)
 10 INPUT "NAME"; N$
 15 PRINT
 20 PRINT "TEST SCORES"
 30 PRINT "FOR "; N$; "==>"
 40 LET QT=0
 50 FOR I = 1 TO 3
 60   PRINT "QUIZ #";I;
 70   INPUT Q
 80   LET QT = QT + Q
 90 NEXT I
100 LET AVE = INT(QT / 3 + .5)
110 INPUT "FINAL EXAM"; F
120 PRINT
130 PRINT "QUIZ AVERAGE =";AVE
140 PRINT "FINAL EXAM   ="; F
145 PRINT
150 PRINT "      ** "; N$
160 PRINT "      HAS ";
170 IF AVE >= 75 AND F>= 70 THEN PRINT "PASSED"
190 IF AVE < 75 OR F < 70 THEN PRINT "FAILED"
200 PRINT : PRINT
210 INPUT "ANOTHER STUDENT"; A$
220 IF LEFT$(A$,1)="Y" GOTO 5
230 END
```

Figure A.4: AND—Sample Program

Notes and Comments

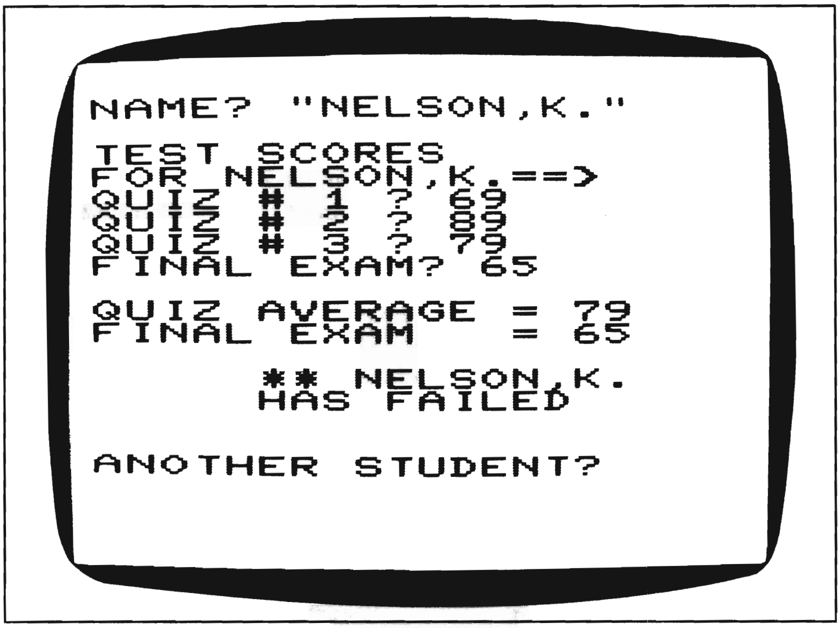
- Figure A.7 is a "truth table" for AND conditions. It shows the resulting value of a compound expression, given different values for statement-1 and statement-2. Notice that the compound statement is true only in one case—when *both* of the inner statements are true.
- Compound statements may consist of more than two logical statements. You can use parentheses to specify the order in which the statements are to be evaluated; for example, if the teacher wanted to allow for a class project, line 170 could be changed to read:

```
IF F >= 70 AND (AVE >= 75 OR PROJECT >= 80) THEN
PRINT "PASSED"
```



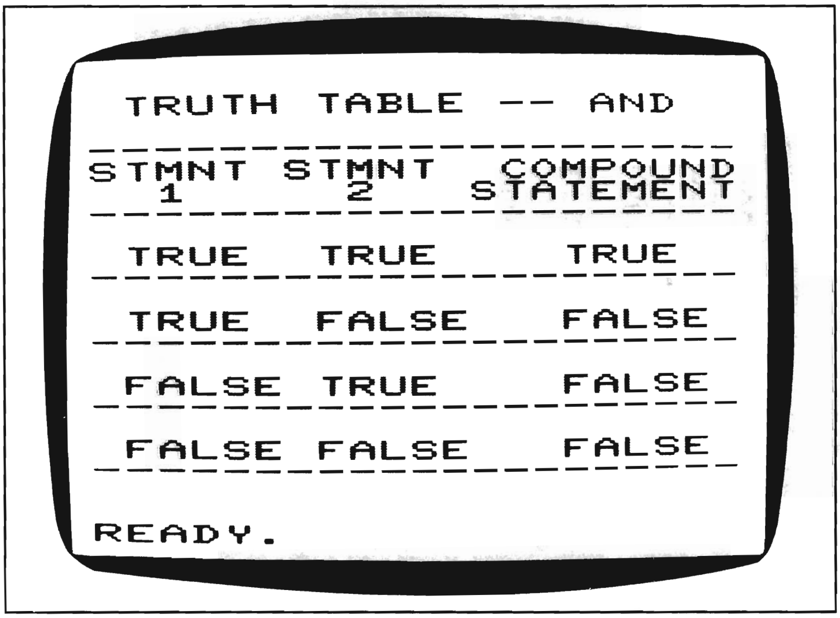
```
NAME? "MARTIN,J."

TEST SCORES
FOR MARTIN,J.==>
QUIZ # 1 ? 85
QUIZ # 2 ? 95
QUIZ # 3 ? 75
FINAL EXAM? 82

QUIZ AVERAGE = 85
FINAL EXAM   = 82

      ** MARTIN,J.
      HAS PASSED

ANOTHER STUDENT?
```

Figure A.5: AND—Sample Output (Compound statement is true.)



```
NAME? "NELSON,K."

TEST SCORES
FOR NELSON,K.==>
QUIZ # 1 ? 69
QUIZ # 2 ? 89
QUIZ # 3 ? 79
FINAL EXAM? 65

QUIZ AVERAGE = 79
FINAL EXAM   = 65

        ** NELSON,K.
        HAS FAILED

ANOTHER STUDENT?
```

Figure A.6: AND—Sample Output (Compound statement is false.)



```
  TRUTH TABLE -- AND

________________________________
STMNT  STMNT   COMPOUND
  1      2    STATEMENT
________________________________

 TRUE   TRUE       TRUE
________________________________

 TRUE   FALSE      FALSE
________________________________

 FALSE  TRUE       FALSE
________________________________

 FALSE  FALSE      FALSE
________________________________

READY.
```

Figure A.7: AND—Truth Table

The compound logical statement in this IF decision would be evaluated to true if and only if both of the following conditions were met:

1. The variable F contains a value that is greater than or equal to 70, *and*
2. At least one of the following statements is true: AVE is greater than or equal to 75, and/or PROJECT is greater than or equal to 80.

— For more information on compound logical statements and IF decisions, *see also* IF, NOT, and OR.

Argument (general programming vocabulary)

An argument is a value sent to a *function*. The function uses the argument in some specified operation, and then returns another value. In most versions of BASIC, the argument appears in parentheses after the name of a function:

```
NAME(ARGUMENT)
```

An argument might be either a numeric or a *string* value, depending on the nature of the function; for example:

```
INT(57.31)
LEN("COMPUTER")
```

Some functions require more than one argument:

```
MID$(S$, 3, 5)
```

In general, the argument of a function may be expressed as a *literal value*, a *variable*, or an expression; for example:

```
COS(3.14)
COS(PI)
COS(PI*2)
```

In the latter two examples, PI is a variable that would have to be assigned a value at some time before the function call.

Arithmetic Expression (programming vocabulary)

An arithmetic expression is one that consists of one or more elements that the computer can evaluate successfully to a single numeric value.

Arithmetic expressions may include *literal* numeric *values, variable* names (which represent the numeric values stored under those variables), *functions*, and operations. The arithmetic operations are represented by the following symbols:

^ exponentiation
* multiplication
/ division
+ addition
- subtraction

The established order of operations in arithmetic expressions is as follows: exponentiation; multiplication and division (from left to right); addition and subtraction (from left to right). To define a different order you may include parentheses in an arithmetic expression.

Array (computer vocabulary)

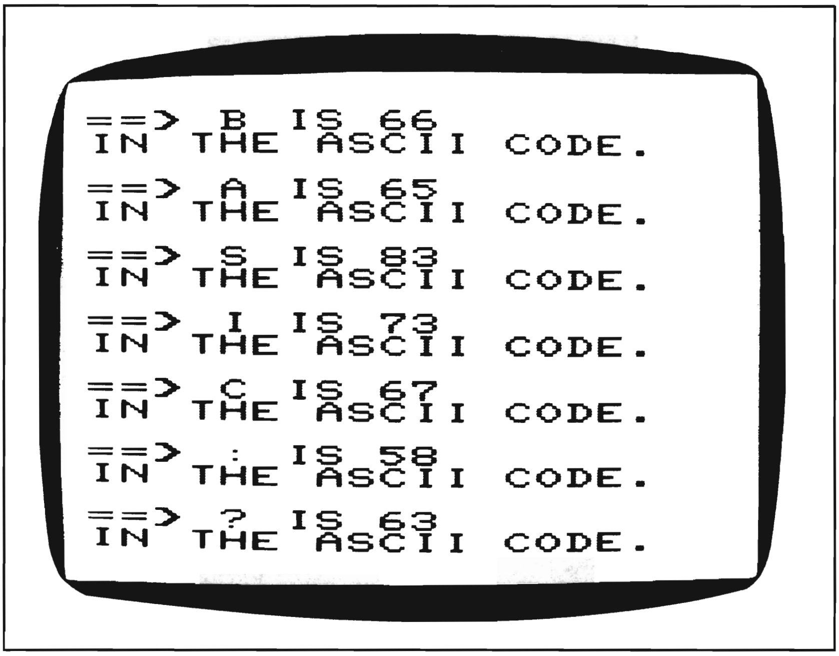
An array is a data structure defined for the storage of lists or tables of data. The name, type, length, and number of dimensions in an array are all defined in a DIM statement. Individual data elements stored in an array are assigned or accessed via an "index" into the array. (*See also* DIM.)

ASC (function)

The ASC *function* is the reverse of CHR$. ASC accepts a single character as its *argument* and supplies the ASCII code number (from 0 to 255) of that character. (If the argument of ASC is a *string* longer than one character, ASC returns the ASCII code of the first character in the string.)

```
10 PRINT CHR$(147)
20 GET I$
30 IF I$="" GOTO 20
40 PRINT "==> ";I$;" IS"; ASC(I$)
50 PRINT "IN THE ASCII CODE."
60 PRINT
70 GOTO 20
```

Figure A.8: ASC—Sample Program



```
==> B IS 66
IN THE ASCII CODE.
==> A IS 65
IN THE ASCII CODE.
==> S IS 83
IN THE ASCII CODE.
==> I IS 73
IN THE ASCII CODE.
==> C IS 67
IN THE ASCII CODE.
==> : IS 58
IN THE ASCII CODE.
==> ? IS 63
IN THE ASCII CODE.
```

Figure A.9: ASC—Sample Output

Sample Program

The program in Figure A.8 demonstrates the use of ASC. The program reads a character from the keyboard (via the GET statement in line 20) and then prints the code number of the character. Line 40 prints the character and the code number:

```
40 PRINT "= => "; I$; " IS"; ASC(I$)
```

The program forms an endless loop (line 70), allowing you to examine as many codes as you wish. Figure A.9 shows a sample run of the program.

Notes and Comments

— *See also* CHR$, for information about the ASCII character code as used on the Commodore computers.

ATN (function)

The ATN *function* supplies the arctangent of any negative or positive *argument*. (The arctangent of a number *x* is defined as the angle whose tangent is *x*.) The result of the ATN function is expressed in radians.

Sample Program

The program in Figure A.10 illustrates ATN. The FOR loop in lines 60 to 80 displays a series of ATN values. Line 70 calls the function itself. The output from this program appears in Figure A.11. Notice that the arctangent of 0 is 0. The result of ATN approaches $+\pi/2$ as the argument approaches infinity; likewise, the result of ATN approaches $-\pi/2$ as the argument approaches minus infinity.

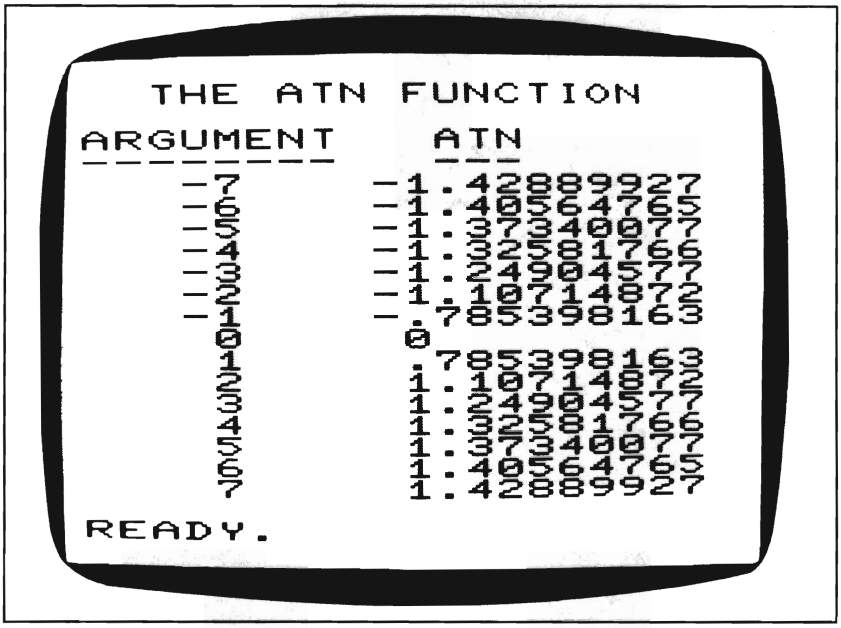
Notes and Comments

— Figure A.12 shows a crude plot of the ATN function, produced on the VIC-20 computer. Mathematically, the arctangent function is called a "multi-valued function," since for any value of *x* (the argument of the function) we can find multiple values of *y* (the result). The portion of the graph from $y = -\pi/2$ to $y = +\pi/2$ (as shown in Figure A.13) is called the "principal branch" of the function. The ATN function returns values from this principal branch.

— To convert from radians to degrees, note that 180 degrees is equal to π radians. Thus, 1 radian is equal to $180/\pi$, or approximately 57.3, degrees.

```
10 PRINT CHR$(147)
20 PRINT TAB(2);"THE ATN FUNCTION"
30 PRINT
40 PRINT"ARGUMENT   ATN"
50 PRINT"--------   ---"
60 FOR I = -7 TO 7
70   PRINT TAB(3);I; TAB(9); ATN(I)
80 NEXT I
```

Figure A.10: ATN—Sample Program



```
   THE ATN FUNCTION
ARGUMENT      ATN
--------      ---
    -7      -1.42889927
    -6      -1.40564765
    -5      -1.37340077
    -4      -1.32581766
    -3      -1.24904577
    -2      -1.10714872
    -1      -.785398163
     0       0
     1       .785398163
     2       1.10714872
     3       1.24904577
     4       1.32581766
     5       1.37340077
     6       1.40564765
     7       1.42889927
READY.
```

Figure A.11: ATN—Sample Output

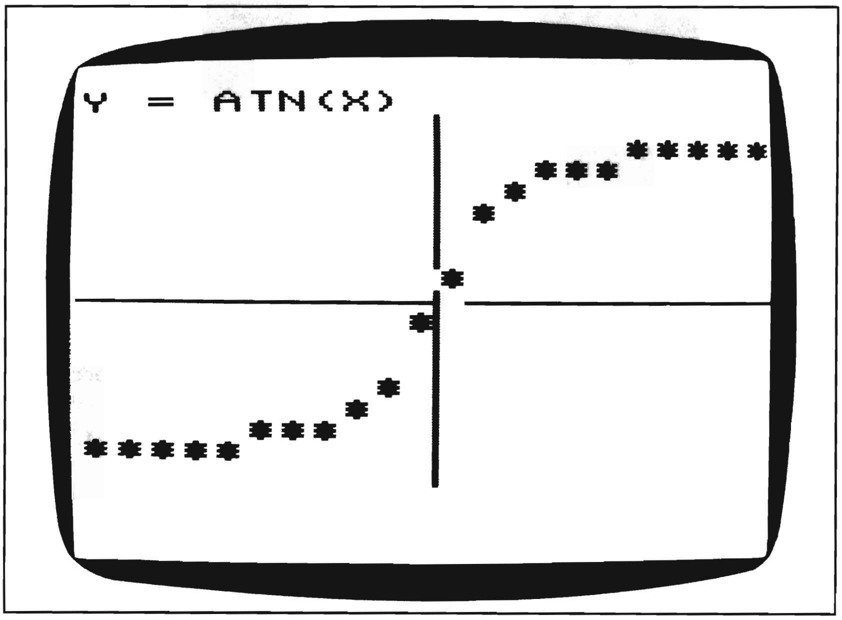


Figure A.12: ATN—Plotted Graph

BASIC (computer vocabulary)

BASIC, which stands for Beginner's All-Purpose Symbolic Instruction Code, is a programming language available on most major microcomputers on the market today. BASIC is characterized by ease of use and a powerful set of instruction commands; however, the plethora of "versions" of the language adds confusion to the situation, and sometimes makes it difficult to transfer a BASIC program from one microcomputer to another. Each version has its own features and liabilities. Compared to other languages (such as Pascal and FORTRAN), BASIC has a limited set of data types and repetition control structures. The Commodore computers' version of BASIC has three data types—integers, floating-point numbers, and *strings;* it also allows typed *arrays.* Like most BASICs, it supplies only two ways of creating repetition loops—the FOR statement, and the GOTO command.

All *variables* are "global" in BASIC; that is, all variables defined in a program are available for use anywhere in the program. There is no facility for creating local *subroutine* variables or for passing values privately from one subroutine to another. (The DEF FN statement might be considered the one exception to this situation. *See the entries under* FN *and* DEF FN.)

Byte (computer vocabulary)

A byte is a unit of memory space, the amount of memory required to store one character. A byte consists of eight "bits" or binary digits. A bit stores one of two possible values—0 or 1. Thus, a byte may hold binary values ranging from:

```
        00000000
```

to:

```
        11111111
```

The decimal equivalent of this range is 0 to 255.

CHR$ (function)

The Commodore computers store their keyboard characters in a numeric code format. The code contains 256 elements; each character is assigned a code number from 0 to 255. The generic name for the code is ASCII, which stands for American Standard Code for Information Interchange. Despite the name, this code is only partly "standard"; each microcomputer manufacturer tends to use it differently, to fit the needs of the specific computer. The Commodore version of ASCII contains letters, digits, punctuation, control characters, and graphics characters. As a result, any of these characters can be stored, in code form, in a single byte of the computer's memory.

The CHR$ *function* supplies the character corresponding to a given ASCII code number. The *argument* of CHR$ must be a code number from 0 to 255; the result is the character that corresponds to that code number.

Sample Program

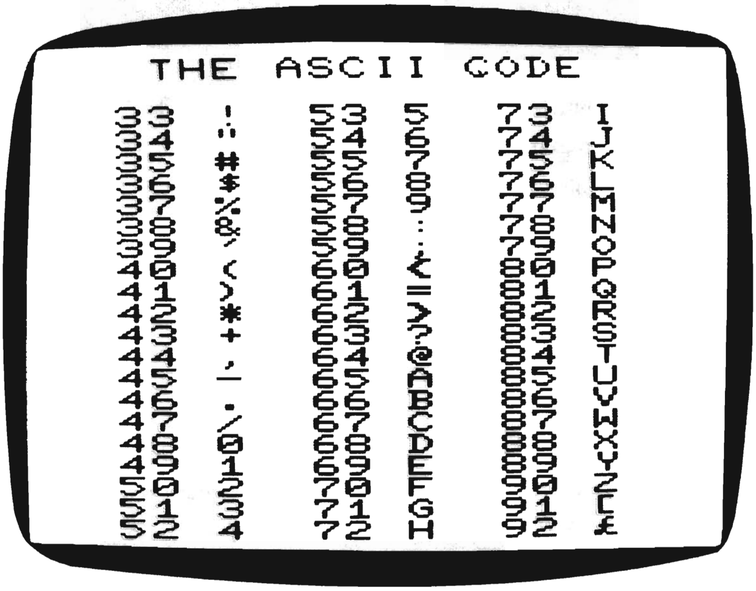
The program shown in Figure C.1 illustrates the use of CHR$ and displays a portion of the ASCII character code on the screen. Figure C.2 shows the output from the program. The symbols, digits, and letters in this portion of the code tend to be fairly standard from one version of ASCII to another.

Lines 40 to 90 of the program form a pair of nested FOR loops that display the codes. The variable C, whose value is calculated from the loop indices I and J, becomes the code number and the argument of CHR$:

```
60 PRINT STR$(C); " "; CHR$(C); " ";
```

```
 10 PRINT CHR$(147)
 20 PRINT TAB(3);"THE ASCII CODE"
 30 PRINT
 40 FOR I = 33 TO 52
 45   PRINT " ";
 50   FOR J = 0 TO 2
 55     LET C=I+J*20
 60     PRINT STR$(C);" "; CHR$(C);" ";
 70   NEXT J
 80   PRINT
 90 NEXT I
100 GET A$
110 IF A$="" GOTO 100
```

Figure C.1: CHR$—Sample Program



```
   THE ASCII CODE

  33 !  53 5  73 I
  34 "  54 6  74 J
  35 #  55 7  75 K
  36 $  56 8  76 L
  37 %  57 9  77 M
  38 &  58 :  78 N
  39 '  59 ;  79 O
  40 (  60 <  80 P
  41 )  61 =  81 Q
  42 *  62 >  82 R
  43 +  63 ?  83 S
  44 ,  64 @  84 T
  45 -  65 A  85 U
  46 .  66 B  86 V
  47 /  67 C  87 W
  48 0  68 D  88 X
  49 1  69 E  89 Y
  50 2  70 F  90 Z
  51 3  71 G  91 [
  52 4  72 H  92 £
```

Figure C.2: CHR$—Sample Output

(Using the STR$ function to convert the code number to a string, while not strictly necessary, makes the output screen easier to plan. *See the entry under* PRINT.)

Notes and Comments

— You can use the CHR$ function, in a PRINT statement, to print graphics characters on the screen or to perform the functions of control characters. The Commodore computers also allow you to type graphics and control characters, between quotation marks, directly into PRINT statements (*see* PRINT); but sometimes using the CHR$ function is more convenient. For example, the first line of the CHR$ program (Figure C.1) clears the screen and positions the cursor at the upper-left corner of the screen (the function of the "HOME" key). The ASCII code of the HOME function is 147:

```
PRINT CHR$(147)
```

Other Commodore ASCII characters that you may want to access via the CHR$ function are the graphics characters (codes 96 to 127, and 161 to 191) and the following control functions:

— switch text display to lower-case: PRINT CHR$(14)

— switch text display to upper-case: PRINT CHR$(142)

— change the color of the text display:

black:	PRINT CHR$(144)
white:	PRINT CHR$(5)
red:	PRINT CHR$(28)
cyan:	PRINT CHR$(159)
purple:	PRINT CHR$(156)
green:	PRINT CHR$(30)
blue:	PRINT CHR$(31)
yellow:	PRINT CHR$(158)

— display characters in reverse video: PRINT CHR$(18)

— move the cursor to the upper-left corner of the screen, without clearing the current display: PRINT CHR$(19)

— move the cursor one position in a specified direction:

up:	PRINT CHR$(145)
down:	PRINT CHR$(17)

```
right: PRINT CHR$(29)
 left: PRINT CHR$(157)
```

— An argument for CHR$ that is outside the legal range (i.e., 0 to 255) will result in the following error message:

```
? ILLEGAL QUANTITY
  ERROR
```

— *See the entry under* ASC for more information.

CLOSE (input/output command word)

The CLOSE command closes an external *file* stored on disk or cassette. The format of CLOSE is:

```
CLOSE F
```

where F is the file number that was specified in the OPEN statement that originally opened the file.

If a tape or disk file is open for writing, the computer automatically completes the writing process before closing the file; this means that any remaining output characters in the file buffer are sent to the file.

CLOSE may be executed as either an *immediate command* or a program statement. *See the entries under* GET#, INPUT#, and PRINT# for sample programs illustrating data file handling techniques; these programs contain examples of the CLOSE command. *See also* OPEN.

CLR (command word)

The CLR command, which may be used either as an *immediate command* or as a program statement, effectively erases the current values of all *variables* and the dimensions of all *arrays*. After CLR, all numeric variables have values of zero, and all *string* variables have null values. Any arrays that you wish to use after CLR must be redefined in a new DIM statement.

Sample Program

The program shown in Figure C.3 is an exercise that demonstrates the effect of CLR. The program displays a series of messages on the screen to tell you what it is doing during the performance, so you can study the results. The first step is to assign random values to each of three variables—X, Y, and Z. This is accomplished in lines 60 to 80; lines 90 to 110 display the variable names and the three values on the screen. The second step is to execute the CLR command, in line 140. Finally, after CLR is performed, an attempt is made to print the three values (of X, Y, and Z) on the screen again, in line 180 of the program.

Figure C.4 shows a run of this program. Study each of the steps. After CLR has been performed, the final value of all three variables is zero.

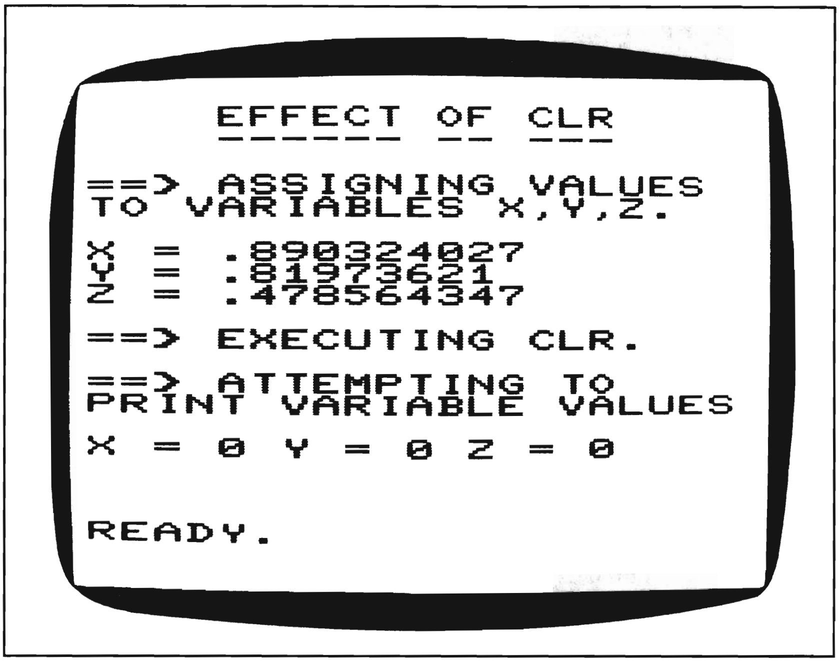
Notes and Comments

— Variables are also cleared under all of the following circumstances:

1. when you use the RUN command to begin a program performance;
2. when you enter the NEW command to clear the current

```
  5 PRINT CHR$(147)
 10 PRINT TAB(4);"EFFECT OF CLR"
 20 PRINT TAB(4);"------ -- ---"
 30 PRINT
 40 PRINT "==> ASSIGNING VALUES"
 50 PRINT "TO VARIABLES X,Y,Z."
 60 LET X=RND(0)
 70 LET Y=RND(0)
 80 LET Z=RND(0)
 85 PRINT
 90 PRINT "X =";X
100 PRINT "Y =";Y
110 PRINT "Z =";Z
120 PRINT
130 PRINT "==> EXECUTING CLR."
140 CLR
150 PRINT
160 PRINT "==> ATTEMPTING TO "
170 PRINT "PRINT VARIABLE VALUES"
175 PRINT
180 PRINT "X =";X;"Y =";Y;"Z =";Z
190 PRINT : PRINT
```

Figure C.3: CLR—Sample Program



```
        EFFECT OF CLR
        ------ -- ---

==> ASSIGNING VALUES
TO VARIABLES X,Y,Z.

X = .890324027
Y = .81973621
Z = .47856434 7

==> EXECUTING CLR.

==> ATTEMPTING TO
PRINT VARIABLE VALUES

X = 0 Y = 0 Z = 0


READY.
```

Figure C.4: CLR—Sample Output

program from memory;

3. when you revise the current program in any way—that is, by adding, deleting, or editing any line of the program.

To run a program, or part of a program, without clearing variable values from a previous run, you must use GOTO as an immediate command. This will only work, however, if you have not edited the program in any way. (*See the entry under* GOTO.)

— *See the entry under* DIM for information on defining and redefining array variables.

CMD (input/output command word)

You can use the CMD command to direct output to a *file* on a device other than the display screen. The format of CMD is:

CMD F, S$

where F is the file number and S$ is an optional string parameter. The value of S$ will be the first data sent to the file. The CMD command must be preceded by an OPEN command that explicitly opens the file numbered F. Depending on the device number specified in the OPEN command, the output device chosen might be a printer, a modem, or an external storage device—that is, a cassette recorder or a disk drive. Following the CMD command, subsequent PRINT statements send output to the specified output device.

You can also use CMD when you want to send a program listing to some external device. For example, the following commands will send the listing to a printer that is specified as device 4:

```
OPEN 1,4
CMD 1, "PROGRAM NAME"
PRINT:PRINT:LIST
PRINT#1
CLOSE 1
```

All of these commands may be entered as *immediate commands*; they will send the current program (stored in the computer's active memory) to the printer. You should, of course, substitute the actual name of the program for the string in the CMD statement.

The following commands will create a sequential data file, called LISTING, on disk, and send the current program listing to that file:

```
OPEN 1,8,2, "0:LISTING, S, W"
CMD 1, "PROGRAM NAME" : PRINT :LIST
PRINT#1
CLOSE 1
```

(*See also* OPEN, for information about data files.)

Concatenation (computer vocabulary)

Concatenation is the combining of two *strings* to form a third string. The plus symbol (+) is used to represent the operation, as in the following example:

```
LET C$ = "CONCAT" + "ENATION"
```

This LET statement stores the string "CONCATENATION" in the variable C$.

In a string expression, the elements of a concatenation might be represented in a variety of ways, including *literal* string values, string *variables*, and the characters returned by string *functions*; for example:

```
LET N$ = L$ + " " + LEFT$(F$,1) + "."
```

CONT (command word)

The CONT command resumes the performance of a program after an interruption caused by an END or STOP statement, or a keyboard interruption (the STOP key). In any of these cases, CONT resumes execution of the program at the next statement (not necessarily the next line, in the case of a program containing multi-statement lines). If the program was interrupted because of a syntax error or an input error, or if you edit the program in any way after the interruption, CONT does not resume the run; instead you will see the *error message*:

```
? CAN'T CONTINUE
  ERROR
```

COS (function)

Given any angle (negative or positive) expressed in radians, the COS *function* supplies the cosine of the angle.

```
  5 DEF FNR(X)= INT(100*X+.5)/100
 10 PRINT CHR$(147)
 20 PRINT TAB(3); "THE COS FUNCTION"
 30 PRINT
 35 PRINT "   ";
 40 PRINT "ARGUMENT    COS"
 45 PRINT "   ";
 50 PRINT "--------    ---"
 60 PRINT
 70 FOR I = -2 TO 2 STEP 1/4
 75   PRINT "   ";
 80   PRINT I;TAB(9);"*PI";TAB(14); FN R(COS(I*3.1416))
 90 NEXT I
100 GET A$
110 IF A$="" GOTO 100
```

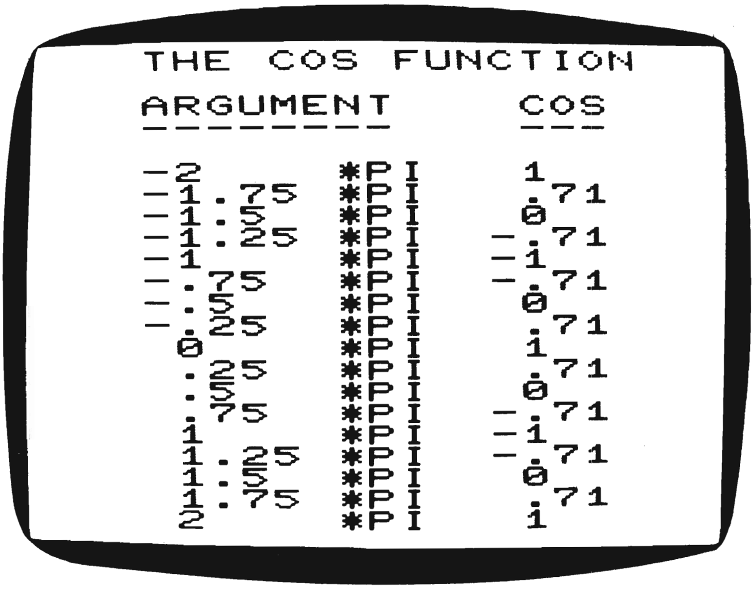
Figure C.5: COS—Sample Program

Sample Program

The program shown in Figure C.5 displays a series of cosine values for *arguments* from -2π to $+2\pi$. The output from this program appears in Figure C.6.

Notes and Comments

- Figure C.7 shows a crude graph of the cosine function, from $x = -2\pi$ to $x = +2\pi$. This graph was created on the VIC-20 computer.
- The other trigonometric functions available in Commodore BASIC are SIN and TAN; the inverse trigonometric function ATN is also implemented.



```
THE COS FUNCTION
ARGUMENT        COS
--------        ---

-2       *PI    1
-1.75    *PI    .71
-1.5     *PI    0
-1.25    *PI   -.71
-1       *PI   -1
-.75     *PI   -.71
-.5      *PI    0
-.25     *PI    .71
 0       *PI    1
 .25     *PI    .71
 .5      *PI    0
 .75     *PI   -.71
 1       *PI   -1
 1.25    *PI   -.71
 1.5     *PI    0
 1.75    *PI    .71
 2       *PI    1
```

Figure C.6: COS—Sample Output

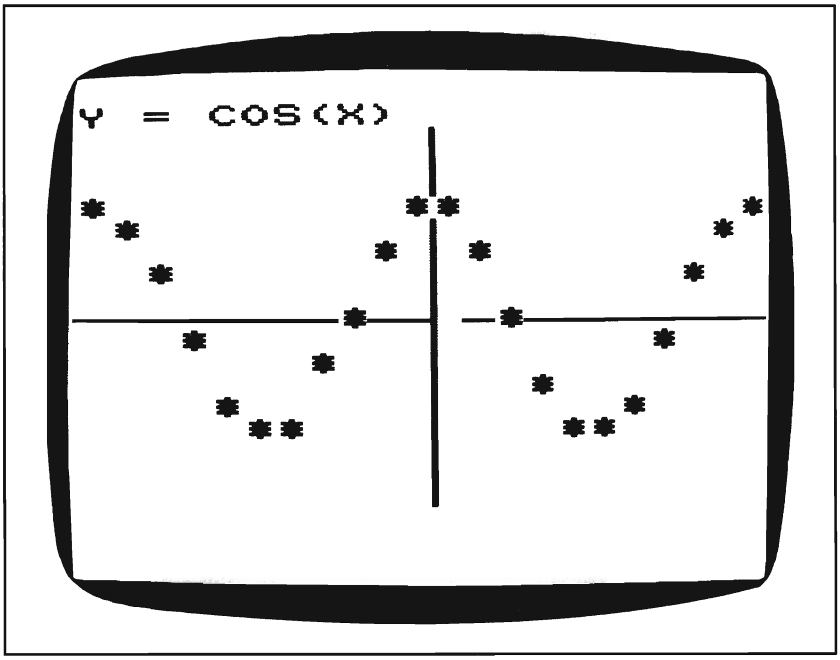
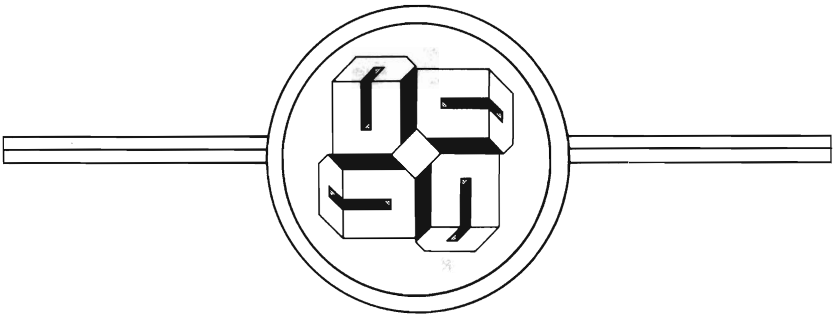


Figure C.7: COS—Plotted Graph

Cursor (computer vocabulary)

The cursor is the small flashing rectangle that appears on the display screen; it indicates the current screen location. Any information sent to the screen will appear starting from this current position. BASIC has some commands and functions that may help you control the position of the cursor at any given point in a program performance. (*See* PRINT.)



DATA (data storage statement)

The DATA statement makes it possible to store a sequence of numeric or *string* data items in a BASIC program. The program can access these data items via the READ command. In some circumstances, the READ/DATA configuration is a simpler and more convenient means of storing and reading data than the creation of an external data *file* on disk or cassette tape.

Any number of DATA statements may be placed at any location in a program listing. The items stored in a DATA statement are separated by commas. A DATA statement may contain numeric data items:

```
100 DATA 10,15,17,23,39
```

or string data items:

```
110 DATA MONDAY, TUESDAY, WEDNESDAY
```

or a combination of both:

```
120 DATA JANUARY, 10, 16, 18.2
```

The number of data items stored in any given DATA statement in a program may vary. (Notice that the statements above contain five, three, and four items, respectively.)

A string data item in a DATA statement may appear with or without surrounding quotation marks. However, if the string contains one or more commas that are intended as part of the data item itself, the quotation marks are required:

```
130 DATA "$1,527,631.81", "$776,821.91"
```

Notice that the comma that separates one string data item from the next must appear outside the quotation marks.

All the DATA statements of a given program together form, in effect, a single sequential data file—that is, a group of data items that can be accessed one at a time in the order in which they appear in the file. (Remember, though, that the DATA statements store this file inside the BASIC program itself, not on any external medium such as a disk or cassette tape.) The computer automatically sets up a "pointer" that keeps track of the current data item in the group of DATA statements. A READ statement accesses the current data item, and causes the computer to increment the pointer to the next data item in the "file." The RESTORE command resets the pointer back to the very first data item. (*See the entries under* READ and RESTORE for more details.)

Sample Programs

It is often convenient to use the READ and DATA statements to assign values to an *array*; such an array might store specific string or numeric data items that you know will be needed each time the program is performed. The data can be read inside a FOR loop. For example, consider the following sequence:

```
10 DIM M$(12)
20 FOR I = 1 TO 12
30    READ M$(I)
40 NEXT I
50 DATA JAN, FEB, MAR, APR, MAY, JUN
60 DATA JUL, AUG, SEP, OCT, NOV, DEC
```

These lines store abbreviations for the names of the months in the string array M$. After the sequence is performed, this array of strings is available for use anywhere in the program. For example, the following line prints the word SEP on the screen:

```
PRINT M$(8)
```

Of course, an alternative approach would have been to use assignment statements to create the array of month names:

```
10 DIM M$(12)
20 LET M$(1) = "JAN"
30 LET M$(2) = "FEB"
40 LET M$(3) = "MAR"
```

... and so on. All and all, however, the READ/DATA approach seems simpler and more economical when more than about ten data items have to be assigned to the elements of an array.

Another important technique that you can use for assigning data to an array is to reserve a special DATA statement to indicate how many data items there are to read. For example, suppose you are writing a program to process data about a staff of salespeople. You want to store the employees' names in the array NA$. At the time you are writing the program, there are fifteen salespeople. You might write the following sequence:

```
10 READ N
20 DIM NA$(N)
30 FOR I = 1 TO N
40    READ NA$(I)
50 NEXT I
60 DATA 15
70 DATA ALCOTT, BAKER, CODY, DUVALL, EVERETT
80 DATA GASSET, HINEMAN, JONES, LANE, NORTON
90 DATA RASTON, SMITH, SUMMERS, WARREN, WHITE
```

Notice that the DATA statement in line 60 contains one item, the number 15. Line 10 of the program reads this number into the variable N, and then line 20 uses N to define the length of the array NA$. The rest of the DATA statements contain the names of the fifteen salespeople; these items are read into the array by the FOR loop in lines 30 to 50. (Notice that the FOR loop also uses the variable N to determine the number of repetitions of the loop. *See the entry under* FOR.)

If at some point in the future the number of salespeople changes, the program can easily be revised to fit the new situation. For example, if five more people are hired, resulting in a total staff of twenty salespeople, you would change line 60 to:

```
60 DATA 20
```

and then simply write additional DATA statements for the names of the new salespeople.

Notes and Comments

— *See the entries under* DIM and RND, for additional illustrations of the DATA statement.

DEF FN (function definition statement)

With the DEF FN statement you can define your own arithmetic *functions* for use in BASIC programs. Defining such a function requires that you specify the three distinct elements of the DEF FN statement:

1. a name for the function you are going to define;
2. a *variable* for use in the definition of the function;
3. an *arithmetic expression* that represents the function's actual calculations.

We might state the general form of the DEF FN statement as follows:

```
DEF FN A(B) = arithmetic expression
```

where A represents the name of the function itself, and B represents a variable name. The arithmetic expression following the equal sign will most likely contain at least one reference to the variable B. A "call" to this function will take the form:

```
FN A(V)
```

where V is a *literal value*, a variable, or an arithmetic expression. This function call results in the following actions:

1. V is evaluated, and its value is "sent" to the variable B in the function definition.
2. The function's arithmetic expression is performed, using the value sent to B.
3. The result of the arithmetic expression is returned as the value of the function.

Consider an example. Let's say you are writing a program in which you frequently have to perform the following operation on some given number:

Multiply the number by itself and add 9 to the result.

Of course, you could simply write a new, but similar, arithmetic expression each time this operation must be performed; but creating a user-defined function is more economical.

We'll name this function S9 (for "square plus 9"). To define the function we'll use the variable X. The function's arithmetic expression will be:

```
X * X + 9
```

Putting together the three elements of the function definition—the name, the variable, and the arithmetic expression—we have:

```
DEF FN S9(X) = X * X + 9
```

Paraphrased, this function definition says, "Store in the variable X the value received from the function call. Multiply X by itself and add 9. Return the result as the value of the function."

An example of a statement that calls this function is:

```
PRINT FN S9(5)
```

This statement sends the value 5 to X in FN S9 and PRINTs the result on the screen. The result displayed will be 34. You can see that FN S9 has indeed performed the calculation specified in the function definition (5 × 5 + 9 = 34).

The value 5 in this example is called the *argument* of the function. The argument is the value that is sent to the function for use in the specified calculation. The argument can also be a variable or even an arithmetic expression, as shown in the following two statements:

```
PRINT FN S9(M)
PRINT FN S9((M + N) * 2)
```

In these statements the variables M and N have values of their own. However the argument is expressed, it is evaluated and sent to the function.

The variable X in the definition of the function FN S9 is sometimes called a "dummy" variable. It is only defined for private use in the function itself. In fact, a variable elsewhere in the program may also be named X; this variable's value will be completely independent of, and remain totally unchanged by, the activities of FN S9. (*See the entry under* BASIC for a discussion of local and global variables.) You can demonstrate this by running the following short program:

```
10 DEF FN S9(X) = X * X + 9
20 LET X = 15
30 LET Y = FN S9(5)
40 PRINT X, Y
```

Line 10 defines the function. Lines 20 and 30 assign values to the variables X and Y; specifically, X is assigned the value 15, and Y receives the value returned from a call to the function FN S9. Line 40 prints the values of both variables; the result on the screen is:

```
15            34
```

From this output you can see that the value of the program variable X is not affected by the call to the function FN S9. The function's dummy variable X is distinct and separate from the variable in the program itself.

Notes and Comments

— The DEF FN statement may not be written as an *immediate command.* However, once a function is defined, a call to the function may be part of an immediate command.

— *See the entry under* FN for a sample program using the DEF FN statement.

DIM (array definition statement)

The DIM (for "dimension") statement allows the programmer to define an *array* and to specify the characteristics of that array. (DIM may be used as either an *immediate command* or a program statement.)

A variable is a place set aside in the computer's memory for a certain value; an array is a collection of variables that are "indexed" for convenient access. Arrays have dimensions. We sometimes refer to a one-dimensional array as a "list" of variables, and a two-dimensional array as a "table" of variables.

In addition to dimension, each array has a specified name, type, and length. The DIM statement provides a convenient way to define all of these characteristics in one simple program statement. For example, the statement:

```
DIM S(10)
```

defines a one-dimensional array named S. We know the array is one-dimensional, because only one number appears in parentheses after the name of the array. The name itself specifies the type of the array; as with simple variables, the last character of the array name indicates the type of data the array can hold. An array with a name that ends in the character % is defined for storing integers; an array with a name that ends in $ is defined for storing strings. The names of real-number arrays end in a letter or a digit.

The array S, as defined above, is thus an array of real numbers. The *length* of S is 11; that is, S can store up to eleven numbers. We can think of S as a list of eleven numeric variables. The names of these eleven variables are as follows:

S(0)
S(1)
S(2)
S(3)
S(4)
S(5)
S(6)

```
S(7)
S(8)
S(9)
S(10)
```

Like any numeric variable, each of these variables can store one numeric value at a time.

In general, a numeric array defined as:

```
DIM A(N)
```

contains N + 1 elements, because the first element of such an array is A(0).

Once the array S has been defined in a DIM statement, you can use these eleven variables in the same ways that you would use any simple numeric variable: you can assign values to them via LET or INPUT statements; you can display their values on the screen using PRINT statements; or you can include these variables in arithmetic expressions to perform calculations on their values.

The number between parentheses in the name of an array element is called the "index" into the array. This number does not have to be a literal numeric value; it can also be represented by a variable. For example:

```
S(I)
```

Now, as long as the variable I contains a value from 0 to 10 (the range of the array S), S(I) refers to one of the eleven values of the array. You can begin to see why arrays are such a convenient way to store data. With a variable as the array index, you can create a working relationship between an array and a FOR loop, to perform long data-processing tasks in very few program statements. For example, the following three lines could print all eleven of the values stored in the array S on the screen:

```
100 FOR I = 0 TO 10
110    PRINT S(I)
120 NEXT I
```

In this sequence, the FOR loop's control variable, I, also serves as the index into the array S. As the FOR loop increments the value of I from 0 to 10, each value of the array S is accessed and printed on the screen, one by one. (This assumes, of course, that S has been assigned values somewhere earlier in the program.) You will see further examples of arrays and FOR loops in the sample program below.

The DIM statement itself may also have a variable name as the index of the array:

```
40  DIM S(N)
```

In this case, the variable N must be assigned a value before the computer encounters the DIM statement during the program run; for example:

```
30  INPUT "HOW MANY VALUES"; N
40  DIM S(N)
```

The value of N, then, will define the length of the array S.

Arrays may be defined with more than one dimension. An example of a two-dimensional array definition appears as follows:

```
    DIM T(3,4)
```

The *table* of variables represented by the array T is:

```
T(0,0) T(1,0) T(2,0) T(3,0)
T(0,1) T(1,1) T(2,1) T(3,1)
T(0,2) T(1,2) T(2,2) T(3,2)
T(0,3) T(1,3) T(2,3) T(3,3)
T(0,4) T(1,4) T(2,4) T(3,4)
```

Note that the index of both dimensions starts at 0. Often you will find that you have no particular use in your programs for this initial element of the arrays you define. This presents no problem; there is no rule that says you *have* to use every element of an array you define. All the same, it is good to keep in mind that an element zero is available for those occasions when it is useful.

You can define as many arrays as you need for any given program. (The only practical limitation is, of course, the amount of memory you have in your computer.) The syntax of the DIM statement is flexible; you may define several arrays in a single DIM statement:

```
10  DIM A(20), B$(10,10), C%(15)
```

or you may write several DIM statements in the same program:

```
10  DIM A(20)
20  DIM B$(10,10)
30  DIM C%(15)
```

```
 10 PRINT CHR$ (147)
 20 PRINT "  BAR GRAPH PROGRAM"
 30 PRINT
 40 PRINT "ENTER MONTHLY DATA"
 50 PRINT "FOR ONE YEAR."
 55 PRINT
 60 INPUT "TITLE OF GRAPH";T$
 70 INPUT "YEAR";Y
 80 DIM M$(12), A(12)
 90 GOSUB 200
100 GOSUB 300
110 GOSUB 400
120 END
200 REM ** MONTH NAMES
210 FOR I = 1 TO 12
220   READ M$(I)
230 NEXT I
240 DATA JAN, FEB, MAR
250 DATA APR, MAY,JUN
260 DATA JUL, AUG, SEP
270 DATA OCT, NOV, DEC
280 RETURN
300 REM ** INPUT
302 GOSUB 600
304 PRINT "INPUT ONE VALUE FOR"
306 PRINT "EACH MONTH."
308 PRINT
309 LET BIG = 0 : LET T = 0
310 FOR I = 1 TO 12
320   PRINT M$ (I)
330   PRINT "---"
340   INPUT A(I)
345   IF A(I)>BIG THEN LET BIG = A(I)
347   LET T = T + A(I)
350 NEXT I
360 RETURN
```

Figure D.1: DIM—Sample Program

```
400 REM ** BAR GRAPH
410 LET FAC = 16/BIG
420 LET C$ = CHR$(168)
425 GOSUB 600
430 FOR I = 1 TO 12
440   PRINT M$(I);"> ";
450   LET L = INT(A(I)*FAC+.5)
460   FOR J = 1 TO L
470     PRINT C$;
480   NEXT J
490   PRINT
500 NEXT I
510 PRINT
520 GOSUB 700
530 RETURN
600 PRINT CHR$ (147)
610 PRINT T$;" --";Y
620 LET L=LEN(T$)+8
630 FOR I = 1 TO L
640   PRINT "-";
650 NEXT I
660 PRINT
670 RETURN
700 REM ** STATISTICS
710 PRINT "HIGHEST MONTH = ";BIG
720 PRINT "AVERAGE =";INT (T/12)
725 PRINT
730 RETURN
```

Figure D.1: DIM—Sample Program, continued

Sample Program

Figure D.1 shows a BASIC program that reads a series of numeric data items from the keyboard, stores that data in an array, and then produces a bar graph from the data. Specifically, the series of data consists of one item for each month of a given year. The program specifies nothing about the *nature* of the data; the numbers could represent any collection of monthly data—from income to rainfall to bowling scores. This program's input routine performs the following tasks:

1. it supplies an input prompt for each month's data;
2. it reads the data from the keyboard;
3. it stores the data in a convenient and usable form.

These tasks alone supply some good illustrations of the use of arrays and the DIM statement. Once the data is stored, any number of different processing tasks are possible. To illustrate one such task, the program goes on to build a bar graph from the data that is stored in the array.

Near the beginning of the program, two values are read from the keyboard:

```
60 INPUT "TITLE OF GRAPH"; T$
70 INPUT "YEAR"; Y
```

The program displays these two values, T$ and Y, at the top of the screen—once at the beginning of the input routine, and again when the bar graph is produced. The *subroutine* at line 600 displays this title.

The DIM statement follows, defining the two arrays M$ and A:

```
80 DIM M$(12), A(12)
```

The string array M$ will hold abbreviated names of the twelve months. The numeric data for each month will be assigned to the array A. Both arrays will thus hold 12 items, one for each month. (Actually, if we were to use the "zeroth" elements, the arrays could hold 13 items; but this program leaves those elements unused.)

The next three lines of the program make subroutine calls:

```
 90 GOSUB 200
100 GOSUB 300
110 GOSUB 400
```

The subroutine at line 200 initializes the values of the string array M$; the subroutine at line 300 actually reads the input data and assigns the values to the array A. The subroutine at line 400 produces the bar graph. Notice that an END statement appears at line 120, effectively separating the "main program" section from the subroutines. (*See* GOSUB.)

The subroutine at line 200 uses the READ/DATA feature to initialize the array M$. The index into the array is incremented from 1 to 12 by a FOR loop at line 210:

```
210 FOR I = 1 TO 12
220    READ M$(I)
230 NEXT I
```

This is the program's first example of the use of a FOR loop to access the elements of an array; these three short lines efficiently instruct the computer to read a value for each of the twelve elements of M$—from M$(1) to M$(12). The DATA statements in lines 240 to 270 store the twelve strings that are to be read into M$. (Of course, we could have initialized M$ by writing twelve assignment statements, but the READ/DATA method is considerably more efficient. *See the entries under* READ and DATA.)

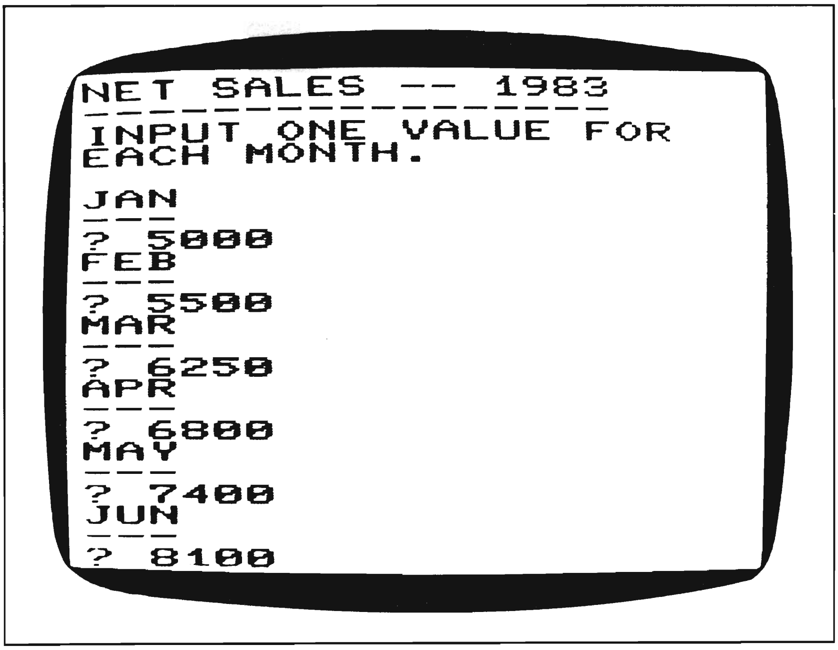
The input routine, at line 300, uses another loop to read and store the values of the array A. The loop begins by displaying a month name on the screen:

```
310 FOR I = 1 TO N
320 PRINT M$(I)
330 PRINT "---"
```

and then reads a numeric data value for that month and stores it in the corresponding element of A:

```
340 INPUT A(I)
```

Figure D.2 shows how this input dialogue looks on the screen.



```
NET SALES -- 1983
-----------------
INPUT ONE VALUE FOR
EACH MONTH.
JAN
---
? 5000
FEB
---
? 5500
MAR
---
? 6250
APR
---
? 6800
MAY
---
? 7400
JUN
---
? 8100
```

Figure D.2: DIM—Input Dialogue

As the input dialogue proceeds, the program determines two statistics about the input data—the largest data item, and the sum of all the items. Line 345 compares each new input value to the current largest value, stored in the variable BIG:

```
345 IF A(I) > BIG THEN LET BIG = A(I)
```

Line 347 accumulates the total of all the items in the variable T:

```
347 LET T = T + A(I)
```

The subroutine at line 400 produces the bar graph. The routine begins by calculating a scale factor, FAC, from the value of BIG:

```
410 LET FAC = 16 / BIG
```

The value 16 represents the maximum length, in columns, available on the screen for one line of the bar graph. (If you have a Commodore 64, you might want to use a greater value here, to take advantage of the 40-character screen display.) In order to determine the length of each line of the graph, we can thus multiply the appropriate data value stored in the array A by the scale factor FAC:

```
    A(I) * FAC
```

This calculation is performed in line 450, resulting in the length, L, of a given line. The bar graph is produced by a pair of nested FOR loops. The outer loop increments the control variable I from 1 to 12 and displays the name of each month, M$(I), on the screen:

```
430 FOR I = 1 TO 12
440 PRINT M$(I); "> ";
```

For each month, the inner loop displays a line that is L screen columns long:

```
460 FOR J = 1 TO L
470    PRINT C$;
480 NEXT J
```

The character stored in C$ is the building-block of the bar graph, the keyboard graphic character located at the left side of the £-key. The ASCII code for this character is 168:

```
420 LET C$ = CHR$(168)
```

Figure D.3 shows a sample bar graph produced by this program.

Notes and Comments

— The elements of all numeric arrays are automatically intialized to zero. To see exactly what this means, run the following short program:

```
10 DIM A(100)
20 FOR I = 1 TO 100
30    PRINT A(I); " ";
40 NEXT I
```

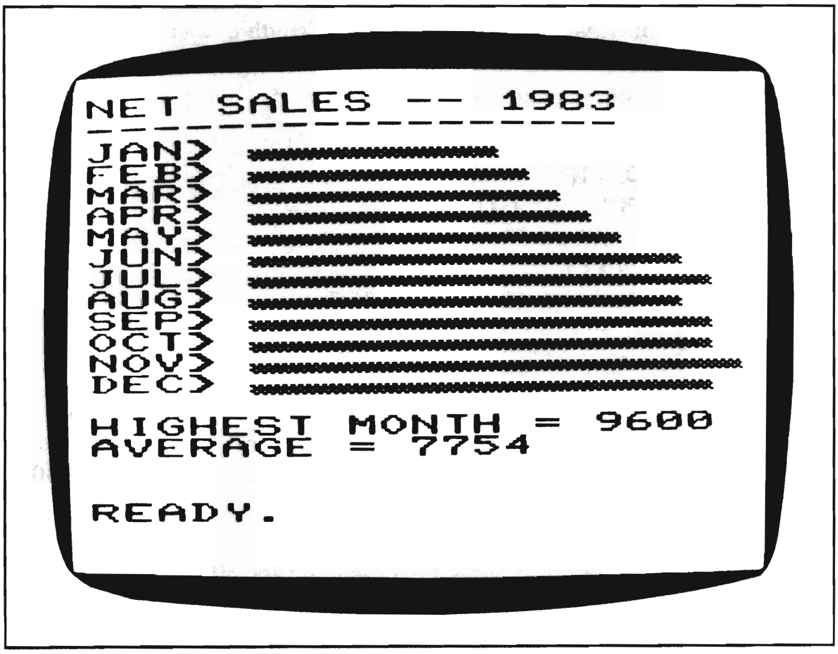
This program will display a series of 100 zeros on the screen, the initial values of the array A.

The elements of all string arrays are initialized to the null string.

— You may use small arrays without first defining them in a DIM statement. For example, consider the following statement:

```
10 LET M(5) = 29
```

If this instruction is *not* preceded by a DIM statement, BASIC



```
NET SALES -- 1983
-----------------
JAN>  ************************
FEB>  ***************************
MAR>  ******************************
APR>  *********************************
MAY>  ************************************
JUN>  ******************************************
JUL>  *********************************************
AUG>  ******************************************
SEP>  *********************************************
OCT>  *********************************************
NOV>  ************************************************
DEC>  *********************************************
HIGHEST MONTH = 9600
AVERAGE = 7754

READY.
```

Figure D.3: DIM—Sample Output

will automatically dimension the array M to a length of 11. In other words, there is an implied DIM statement, as though you had actually written the following two commands:

```
10  DIM M(10) : LET M(5) = 29
```

— If you try to use arrays longer than 11 elements (i.e., with elements numbered 0 to 10) without first defining them in a DIM statement, your program will terminate with the following *error message*:

```
?  BAD SUBSCRIPT
   ERROR
```

Even though the automatic dimension feature can be used for small arrays, you should get into the habit of writing DIM statements for all arrays, whatever their length. Not doing so can lead at best to confusion, and at worst to programs that fail.

— BASIC does not allow you to define the same array more than once in a program unless you first give the CLR command. This can be an issue whenever you want to use the same array to store several different sets of data during the same program run. If you intend to change the length of an array, you must first execute a CLR statement.

For example, consider the following short program:

```
10  INPUT "HOW MANY ITEMS"; N
20  DIM T(N)
30  FOR I = 1 TO N
40     INPUT T(I)
50  NEXT I
60  REM ** PROCESS THE DATA
70  CLR
80  GOTO 10
```

This program might represent a situation in which you need to process many lists of data in a certain way. The number of items in any given list is defined in lines 10 and 20; lines 30 to 50 read the data. Line 60 could be replaced by a GOSUB statement that calls the data-processing subroutine. Finally, before the program processes another list (line 80), line 70 clears the previous definition of the array T. Try running the program; you will see that you can define and process as many lists of data as you want. Now delete line 70, the CLR command, and run the program again. As soon as you try to process a second list of data, the program run will terminate with the following error message:

```
? REDIM'D ARRAY
  ERROR IN 20
```

This means that the computer has encountered the DIM statement for T (at line 20) a second time, with no CLR statement intervening.

— Each element of a string array is of "dynamic" length. For example, consider the array defined as follows:

```
  DIM S$(10)
```

S$ can hold up to 11 strings, and the length of each string can be different. In other words, each element of S$ can contain a different number of characters.

— If you try to access an array element that is outside the defined length of the array, your program will terminate with an error message:

```
? BAD SUBSCRIPT ERROR
```

END (command word)

END marks the final line of a BASIC program run. When the computer encounters the END command, it stops performance of the BASIC program and returns control to the command level of the system. In many programs END is an optional command; the computer also terminates a program performance when it simply runs out of lines to perform. However, when a program is organized modularly—with a "main program" section that "calls" the *subroutines* below it—an END or STOP statement is required to separate the main program from the subroutines. (*See the entry under* GOSUB.)

Notes and Comments

- The STOP command also halts a program run, and supplies a BREAK message on the screen to tell you exactly where the program ended. (*See* STOP.)
- The CONT command causes the computer to resume the performance of a program at the line following the END or STOP statement that halted a performance. (*See* CONT.)

Error Message (computer vocabulary)

If, during the course of a program performance, the computer encounters some error situation that it cannot deal with in a normal way, it interrupts the program performance and displays an error message on the video screen. In the best of all possible worlds, every error message would express unambiguously the nature and, when appropriate, the location of the error. Unfortunately, this cannot always be the case; you will sometimes be left with some detective work to find out exactly what went wrong in a program. Examples of error messages are given thoughout this book.

EXP (function)

The EXP function supplies the natural exponent of a number; that is, a power of *e*, where *e* = 2.71828183. The expression:

EXP(V)

means *e* to the power of V.

Sample Program

The program in Figure E.1 is designed to display a range of values from the EXP function, for a series of negative and positive arguments. The output from the program appears in Figure E.2.

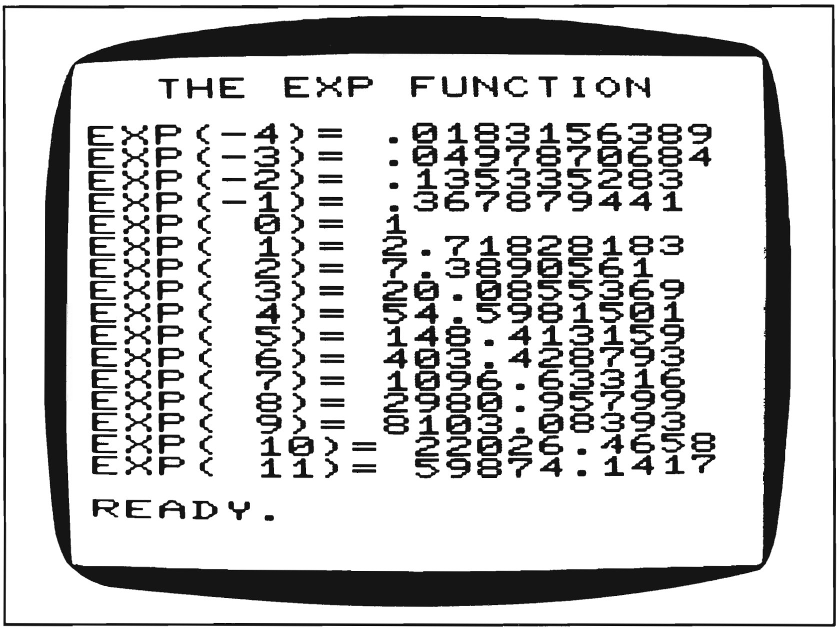
Note that as the argument increases in the negative direction, the value returned by EXP moves closer and closer to zero.

Notes and Comments

— Figure E.3 shows a plotted graph of the EXP function.

```
 5 PRINT CHR$(147)
10 PRINT "  THE EXP FUNCTION"
20 PRINT
30 FOR I = -4 TO  11
40   PRINT "EXP(";STR$(I);")";
50   PRINT "="; EXP(I)
60 NEXT I
```

Figure E.1: EXP—Sample Program



```
     THE EXP FUNCTION
EXP(-4)=  .0183156389
EXP(-3)=  .0497870684
EXP(-2)=  .135335283
EXP(-1)=  .367879441
EXP( 0)=  1
EXP( 1)=  2.71828183
EXP( 2)=  7.3890561
EXP( 3)=  20.0855369
EXP( 4)=  54.5981501
EXP( 5)=  148.413159
EXP( 6)=  403.428793
EXP( 7)=  1096.63316
EXP( 8)=  2980.95799
EXP( 9)=  8103.08393
EXP( 10)=  22026.4658
EXP( 11)=  59874.1417
READY.
```

Figure E.2: EXP—Sample Output

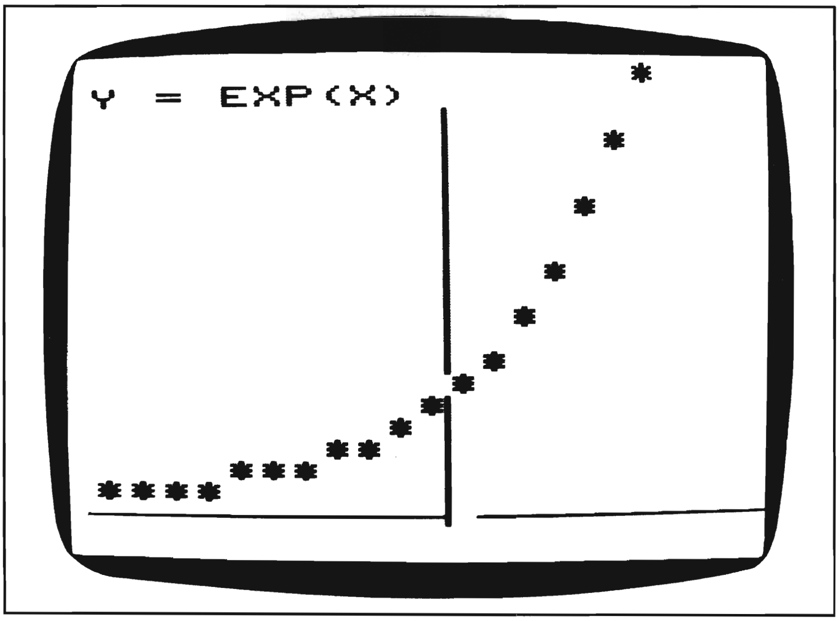


Figure E.3: EXP—Plotted Graph

File (computer vocabulary)

In the context of tapes and disks, a file is a collection of information stored under a given name. A set of commands is available for creating and maintaining such external files. The format of these commands differs, depending on whether you are using a cassette recorder or a disk drive to store files. The SAVE and LOAD commands are for saving and retrieving BASIC program files. The OPEN, CLOSE, PRINT#, INPUT#, and GET# commands are for creating and reading data files. A file name may be from 1 to 16 characters long.

BASIC also allows you to create a kind of data file stored within the program listing itself. The DATA statement is designed for the storage of the data items, and the READ statement is available for accessing the data for use in the program. In addition, the RESTORE command allows you to access the data, from the beginning of the "file," as many times as necessary. (*See* DATA, READ, and RESTORE.)

FN (user-defined function call)

FN designates a call to a user-defined *function*. The call takes the form:

```
FN A(V)
```

where A is the name of a function that has been defined in a DEF FN statement, and the value V is the *argument* that will be sent to the function. V may be a *literal* value, a *variable*, or an *arithmetic expression*. (*See the entry under* DEF FN for further details.)

Sample Program

The program in Figure F.1 is a graph-plotting exercise that illustrates the DEF FN and FN statements in action. The program plots points of the following equation on the screen:

$$x = y^2/4$$

As you can see from the program's output, shown in Figure F.2, the equation defines a parabola. Program line 30 specifies the user-defined function:

```
30 DEF FN X(Y) = Y * Y / 4
```

```
 10 PRINT CHR$(147)
 20 GOSUB 300
 30 DEF FN X(Y)=Y*Y/4
 35 PRINT "PARABOLA"
 40 FOR I = -9 TO 9
 50   PRINT TAB(FN X(I));"*"
 60 NEXT I
 70 GOSUB 500
 80 END
300 FOR I = 1 TO 10
310   PRINT CHR$(221)
320 NEXT I
330 FOR I = 1 TO 21
340   PRINT CHR$(192);
350 NEXT I
355 PRINT
360 FOR I = 1 TO 10
370   PRINT CHR$(221)
380 NEXT I
390 PRINT CHR$(19)
400 RETURN
500 GET A$
510 IF A$="" GOTO 500
520 RETURN
```

Figure F.1: FN—Sample Program

The function is called inside a loop that plots the graph from $y = -9$ to $y = +9$:

```
40 FOR I = -9 TO 9
```

The plotting technique is simple: to find the horizontal location for each point of the graph, the program must TAB forward by the correct number of columns, and then print an asterisk at that location. The value returned by the function FN X thus indicates the correct horizontal location of each asterisk:

```
50 PRINT TAB(FN X(I)); "*"
```

The program also includes two *subroutines*. The subroutine at line 300 draws the *x*- and *y*-axes. Notice the use of the CHR$ function to print graphics characters included in the Commodore version of the ASCII code. The character represented by ASCII code 221 is the building-block for the *y*-axis; code 192 is used for the *x*-axis. (To learn how to draw lines on the screen, *see the entry under* FOR.) Finally, the character represented by

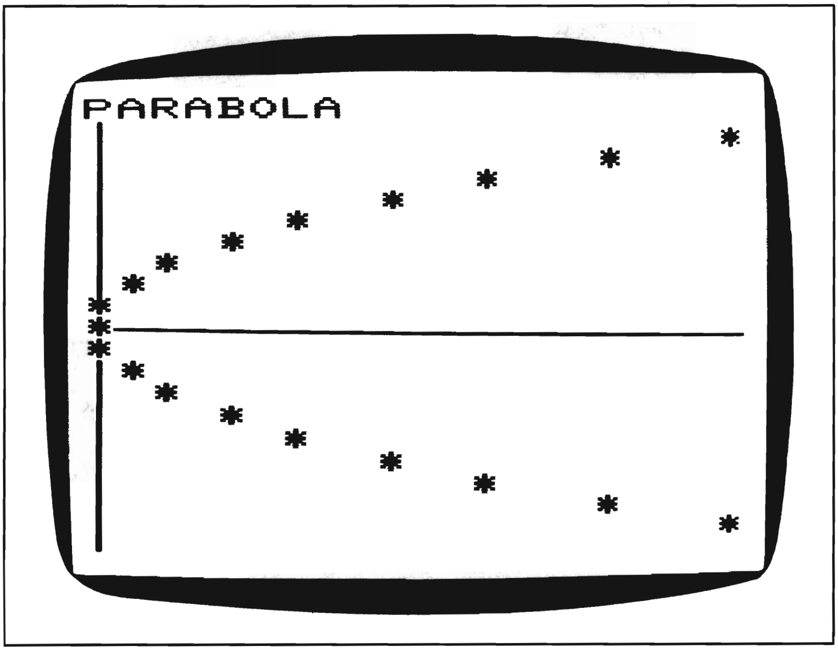


Figure F.2: FN—Sample Output

code 19 repositions the cursor at the upper-left corner of the screen, without clearing the current display:

```
390 PRINT CHR$(19)
```

The subroutine at line 500 simply postpones termination of the program until you press any key on the keyboard.

FOR (command word)

The FOR statement creates a repetition structure, often called a FOR loop. In constructing a FOR loop, the programmer can instruct the computer to repeat the performance of a certain series of lines a specified number of times. An essential element of the FOR loop is that the computer sets up a counting *variable* (called the "control variable"); the value of this variable changes after each repetition of the loop, and the computer uses the variable to determine the number of repetitions.

Two other BASIC words are always part of the FOR loop syntax—TO and NEXT. TO indicates the range of values that the control variable will take during the repetition process. NEXT is a marker for the last line of the FOR loop. Consider this example:

```
 40 FOR I = 1 TO 10
    . . .
100 NEXT I
```

Line 40 specifies that the control variable of this FOR loop is I. The variable I will initially be assigned the value 1, and will be incremented by 1 up to 10 during the course of the repetition process. Line 100 marks the end of the FOR loop with the word NEXT, and refers once again to the control variable, I. Since the variable I will take on 10 different values during the repetition process, the instructions located between line 40 and line 100 will be repeated 10 times, once for each value of I.

Any BASIC instructions may appear as lines inside the FOR loop. Often the lines inside the loop make use of the control variable; for example:

```
 40 FOR I = 1 TO 10
 50    PRINT I
    . . .
100 NEXT I
```

Line 50 would display each value of the control variable on the screen as the repetition proceeds.

BASIC also allows FOR loops to appear inside other FOR loops; such loops are called "nested" FOR loops:

```
 40 FOR I = 1 TO 10
 50    FOR J = 1 TO 5
       . . .
 90    NEXT J
100 NEXT I
```

In this case, the inner loop will go through five repetitions for every repetition of the outer loop. The result will be that lines located between line 50 and line 90 will be performed 50 times (10 × 5) in all.

There are two essential rules to keep in mind while constructing nested FOR loops:

1. The inner loop must have its own control variable, distinct from the control variable of the outer loop. (In the example above, the inner loop's control variable is J.)
2. The inner loop must be completely contained within the outer loop. That is, the FOR and NEXT lines of the inner loop must be located inside the section of the program marked off by the FOR and NEXT lines of the outer loop. The following construction is thus not legal:

```
10 FOR I = 1 TO 10
20    FOR J = 1 TO 5
      . . .
50 NEXT I
60    NEXT J
```

This error will result in the following *error message*:

```
? NEXT WITHOUT FOR
  ERROR IN 60
```

The range of values for the control variable of a FOR loop may be expressed as literal numeric values (as in the example above) or as variables or arithmetic expressions, as in the following line:

```
40 FOR I = A TO B + C
```

The variables A, B, and C must be assigned values before the computer arrives at line 40. The control variable I, then, will be incremented by 1, from A to B + C during the repetition process.

Finally, the amount by which the control variable is incremented for each repetition of the loop can be specified as some value other than 1 through use of the optional word STEP; for example:

```
FOR I = 2 TO 10 STEP 2
```

When STEP does *not* appear in the FOR statement, the default incrementation amount is 1, as we have already seen. (For more examples, *see the entry under* STEP.)

Sample Program

The listing in Figure F.3 shows a classic exercise demonstrating the action of FOR loops. This program creates a multiplication table on the screen, the kind of table that school-children used before the arrival of the pocket calculator.

The two nested loops in lines 30 to 90 create the table. The inner loop, with the control variable J (lines 50 to 70), calculates and prints a single row of values; the outer loop, with the control variable I, carries the process through the ten rows of the table. Line 60, then, is performed 70 times in all, once for each of the 70 entries of the table. Notice that line 60 makes use of the control variables I and J to calculate each entry:

```
I * J
```

The same instruction also uses the control variable J to tab across the screen for the correct placement of each new entry:

```
TAB((J - 1) * 3)
```

Two more FOR loops appear in the program, in the subroutine at line 200. The first of these loops (lines 230 to 250) creates a horizontal line, and the second (lines 270 to 290) creates a vertical line, to mark off the first row and the first column of the table, respectively. Both of these lines are built from keyboard graphics characters included in the Commodore version of the ASCII code. (*See* CHR$.) The character that makes up the horizontal line is ASCII code 192; the line consists of 21 of these characters, printed side-by-side:

```
220 FOR I = 1 TO 21
230     PRINT CHR$(192);
240 NEXT I
```

Notice the semicolon at the end of the PRINT statement in line 230; it prevents a carriage return after each character is printed. The vertical line is built from the ASCII code character 219.

The output from this program appears in Figure F.4.

Notes and Comments

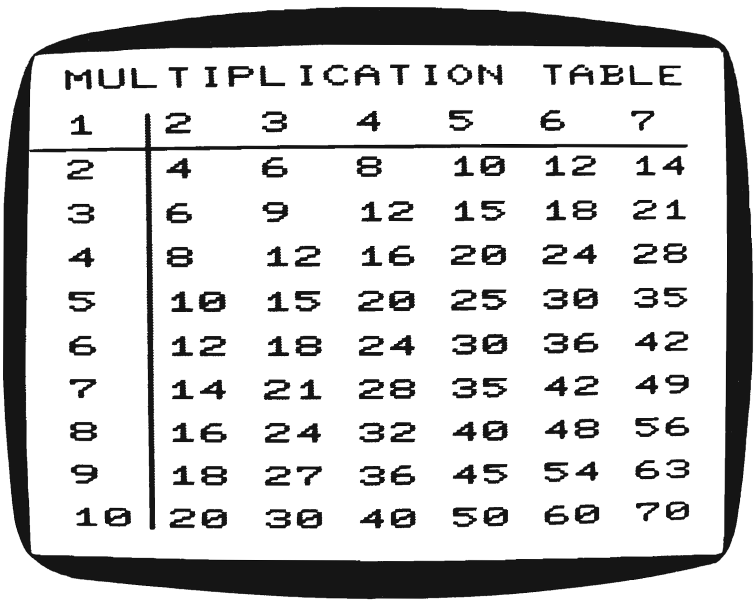
— If the range of the control variable, as specified in the FOR statement, is expressed in the wrong direction, the FOR loop will still perform all of its instructions, but only once. For example:

```
10 FOR I = 10 TO 0
20    PRINT I
30 NEXT I
```

This loop will perform the PRINT statement one time, resulting

```
 10 PRINT CHR$(147)
 20 PRINT " MULTIPLICATION TABLE"
 30 FOR I = 1 TO 10
 40   PRINT
 50   FOR J = 1 TO 7
 60     PRINT TAB((J-1)*3);STR$(I*J);
 70   NEXT J
 80   PRINT
 90 NEXT I
 95 GOSUB 200
100 GET A$
110 IF A$="" GOTO 100
120 END
200 PRINT CHR$(19)
210 PRINT:PRINT:PRINT
220 FOR I = 1 TO 21
230   PRINT CHR$(192);
240 NEXT I
250 PRINT CHR$(19)
260 PRINT:PRINT:PRINT TAB(3); CHR$(221)
265 PRINT TAB(3); CHR$(219)
270 FOR I = 1 TO 17
280   PRINT TAB(3); CHR$(221)
290 NEXT I
300 RETURN
```

Figure F.3: FOR—Sample Program



```
MULTIPLICATION TABLE
1  |2   3   4   5   6   7
---+---------------------
2  |4   6   8   10  12  14
3  |6   9   12  15  18  21
4  |8   12  16  20  24  28
5  |10  15  20  25  30  35
6  |12  18  24  30  36  42
7  |14  21  28  35  42  49
8  |16  24  32  40  48  56
9  |18  27  36  45  54  63
10 |20  30  40  50  60  70
```

Figure F.4: FOR—Sample Output

in an output display of the value 10, the initial value of I; then the loop's action will terminate.

The following loop, however, thanks to a STEP clause, has a decrementing control variable, and will print out the values of the variable from 10 down to 0:

```
10 FOR I = 10 TO 0 STEP -1
20    PRINT I
30 NEXT I
```

See the entry under STEP for additional information about FOR loops.

FRE (system function)

The FRE *function* returns the number of *bytes* of memory still available for a BASIC program. The value of the *argument* of FRE is irrelevant; it has no effect on the result.

The statement:

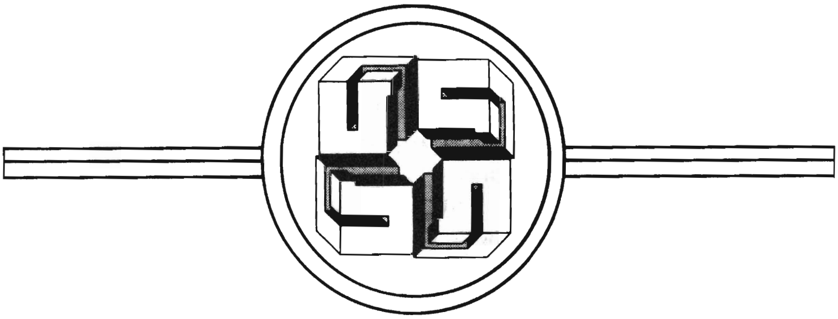
```
PRINT FRE(0)
```

will result in a display of the number of bytes remaining for your program.

Function (computer vocabulary)

A function is a built-in routine that returns a specified type of value, or, in a few cases, performs a screen-display operation. The name of the function itself evokes the routine, and, in *arithmetic* or *string expressions,* stands for the value that the function returns. Functions require *arguments*; an argument is a value that you send to the function to take part in the operation that the function performs. (*See the entry under* Argument.)

BASIC also allows you to define your own arithmetic functions, using the DEF FN statement. (*See the entries under* DEF FN and FN.)



GET (command word)

The GET command reads a single key entry from the keyboard and assigns its value to a *variable.* GET is usually used with a *string* variable; it appears in the following form:

```
10 GET S$
```

This statement causes the computer to scan the keyboard only once in search of an input character. If you have not pressed a key during the instant it takes to scan the keyboard, the GET command assigns a null value to S$—that is, S$ becomes an empty string. For this reason, GET is usually written in a loop to ensure that the computer will continue scanning the keyboard until you do press a key:

```
10 GET S$
20 IF S$ = ""GOTO 10
```

After the GET command in line 10, line 20 tests the value of S$ to see if it is an empty string; if it is, control is returned to line 10 for another search of the keyboard. This loop continues until a key is pressed and its character value is assigned to S$.

As an input command, GET is as important for what it does *not* do as for what it does. GET does not automatically "echo" (i.e., display on the screen) the value of the key you press. This is one of the important distinctions between the GET and INPUT statements. INPUT echoes each key you press on the keyboard by displaying the corresponding character on the screen. GET leaves it to you, the programmer, to decide if and where an input character should be displayed on the screen.

Sample Program

A good programmer always gives careful thought to the way the screen display changes in response to the user's input from the keyboard. The program in Figure G.1, which allows you to change the screen colors—background, frame, and text—quickly and easily, illustrates the use of GET. As you study this program, you will realize that GET is harder to use than the INPUT statement, but that, in return, GET allows some rather elegant input-and-response patterns during a program run.

Look first at what the program does. (To see the action, you should type it into your computer and run it yourself. If you are using a VIC-20 computer, type the program exactly as you see it. If you have a Commodore 64 computer, enter the two POKE commands that appear in the REM statement of line 507 instead of the POKE command in line 500.) The program begins by displaying the following title on the screen:

```
COLOR CHANGER.
<1> TO <8>
```

Following this title, you will see a series of three input prompts appear on the screen, for the background, the frame, and the text displays. For example:

```
BACKGROUND?
```

In response to each prompt, you must press one of the number keys, from 1 to 8. Notice that in front of each of these keys is printed the name of a color; to choose one of these colors, you simply press the corresponding key in response to a prompt. For example, let's say you want to change the background color to yellow. The word YEL is printed in front of the 8-key. If you press the 8-key, you will see the following response appear on the screen:

```
BACKGROUND? YELLOW
```

In addition, the next prompt will appear:

```
FRAME?
```

If, in response to one of the three prompts, you accidentally press a key other than a number key from 1 to 8, there will be no response on the screen. After you have pressed appropriate keys in response to all three prompts, the colors you have requested will appear on the screen. For

```
 10 PRINT CHR$(147)
 20 GOSUB 200
 30 PRINT "COLOR CHANGER."
 40 PRINT "<1> TO <8>"
 45 PRINT
 50 PRINT "BACKGROUND? ";
 60 GOSUB 400:LET B=V
 70 PRINT "FRAME?      ";
 80 GOSUB 400:LET F=V
 90 PRINT "TEXT?       ";
100 GOSUB 400:LET T=V
110 GOSUB 500
120 END
200 FOR I = 1 TO 8
210   READ C$(I),C(I)
220 NEXT I
230 DATA BLACK, 144
240 DATA WHITE, 5
250 DATA RED,28
260 DATA CYAN, 159
270 DATA PURPLE, 156
280 DATA GREEN, 30
290 DATA BLUE, 31
300 DATA YELLOW, 158
310 RETURN
400 GET V$
410 IF V$="" OR V$<"1" OR V$>"8" GOTO 400
420 LET V=VAL(V$)
430 PRINT C$(V)
440 RETURN
500 POKE 36879, 16 * B + F - 9
505 REM COMMODORE 64:
507 REM POKE 53280,F-1: POKE 53281,B-1
510 PRINT CHR$(C(T))
520 RETURN
```

Figure G.1: GET—Sample Program

example, if you press the three keys 1, 5, and 7, you will see:

```
COLOR CHANGER.
<1> TO <8>

BACKGROUND?   BLACK
FRAME?        PURPLE
TEXT?         BLUE

READY.
```

When this dialogue is complete, and the program run is over, the background will be black, the frame purple, and the text blue.

In summary, this program expects exactly three keystrokes of input from you, all three of them numbers from 1 to 8. Significantly, no input outside of this range will produce any response on the screen during the input dialogue. When you press a key that represents a valid input choice, the screen "echo" is a word that tells you what color you have chosen, rather than a number, which would have less meaning on the screen. We will see that this kind of dialogue control is made possible through the GET command.

The program is organized into a "main program" section and three subroutines:

The subroutine at line 200 uses READ and DATA statements to initialize two arrays—C$ stores the names of the eight colors, and C receives the ASCII values of the corresponding color "switches." (*See the entry under* CHR$ for details.)

The subroutine at line 400 is the input routine, which we will look at carefully.

The subroutine at line 500 performs the necessary POKE and PRINT commands to change the colors, after the input dialogue is complete. Notice that for the VIC-20 the background and frame colors are set in a single memory location (36879), whereas the Commordore 64 has two separate locations for these settings (53280 and 53281).

The main program section calls the subroutine at line 400 three times (lines 60, 80, and 100), once for each input value. The routine uses GET to read a character from the keyboard and stores the input character in the variable V$:

```
400 GET V$
```

Line 400 tests the value of V$:

```
410 IF V$ = "" OR V$ < "1" OR V$ > "8" GOTO 400
```

If V$ is empty or contains some character that is outside the correct range—"1" to "8"—control returns to line 400. However, if V$ holds one of the eight valid input characters, the routine continues. The VAL function returns the numeric equivalent of V$, which is assigned to the real variable V:

```
420 LET V = VAL(V$)
```

Finally, V serves as an index into the array C$, to print the name of the chosen color on the screen:

```
430 PRINT C$(V)
```

Upon returning control to the main program section, the program assigns the value of V to one of three "holding" variables—B for background, F for frame, or T for text:

```
 60 GOSUB 400 : LET B = V
    ...
 80 GOSUB 400 : LET F = V
    ...
100 GOSUB 400 : LET T = V
```

The subroutine at line 500 uses these three variables to POKE and PRINT the new screen colors.

GET# (data input command word)

The GET# command reads data from an external file, one character at a time. GET# may be used only as a program statement, not as an immediate command. The format of GET# is:

```
GET#F, V
```

where F is the file number, and V is a variable name. The result of the command is to read one character of data and store it in the variable V. The command also allows a list of variables, in which case one character of data is read for each variable in the list:

```
GET#F, V1, V2, V3, ...
```

String variables are generally safer and more convenient than numeric variables to use in a GET# statement. (If a GET# statement reads a nonnumeric character and attempts to store it in a numeric variable, the program will terminate with a syntax error.)

Sample Program

Figure G.2 shows a program illustrating the GET# command. This program reads the file called EMPLOYEES, created by the sample program listed under the heading PRINT#. The sample program under INPUT# reads this file item-by-item, and produces a table from the data stored in the file. The GET# program does not produce as readable an output screen (Figure G.3), but simply reads and displays each character of the file in sequence.

The OPEN command in line 10 opens the EMPLOYEES file for reading, assigning it the file number 1:

```
10 OPEN 1,8,2, "EMPLOYEES, S, R"
```

Line 20 reads a character of the file and stores it in the variable C$:

```
20 GET#1, C$
```

Lines 30 and 40 perform two tests before displaying the value of C$ on the screen. Line 30 looks at the value of ST (the input/output status function) to see if the end of the file has been reached. The ST value 64 means end-of-file:

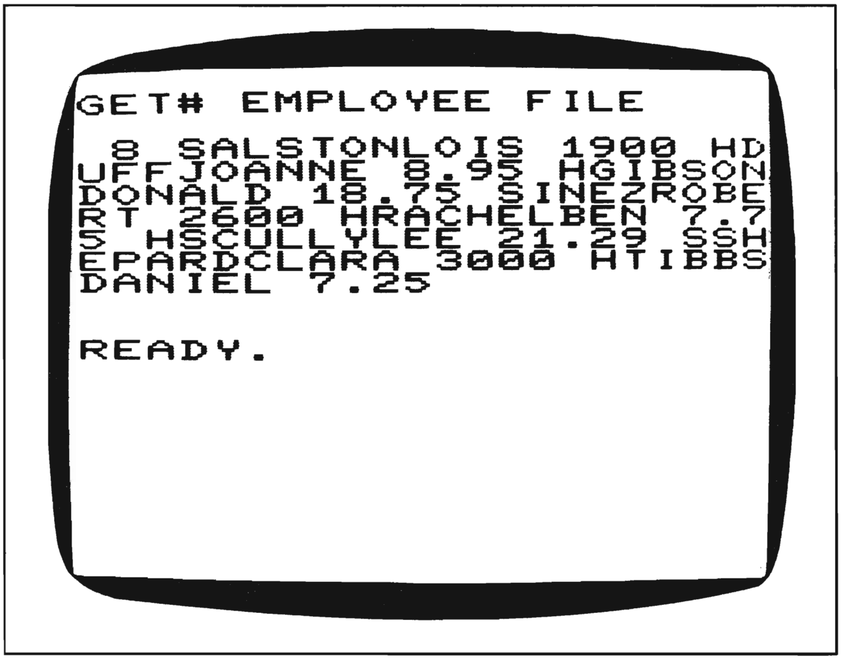
```
30 IF ST = 64 GOTO 60
```

If this condition has been met, control of the program is sent to line 60, which closes the file. Line 40 checks to see if C$ contains a RETURN character, ASCII code 13; if not, the character is displayed on the screen:

```
40 IF C$ < > CHR$(13) THEN PRINT C$;
```

```
 5 PRINT CHR$(147)
 7 PRINT "GET# EMPLOYEE FILE"
 9 PRINT
10 OPEN 1,8,2,"EMPLOYEES,S,R"
20 GET#1,C$
30 IF ST=64 GOTO 60
40 IF C$<>CHR$(13) THEN PRINT C$;
50 GOTO 20
60 CLOSE 1
70 PRINT:PRINT:END
```

Figure G.2: GET#—Sample Program



```
GET# EMPLOYEE FILE
 8 SALSTONLOIS 1900 HD
UFFJOANNE 8.95 HGIBSON
DONALD 18.75 SINEZROBE
RT 2600 HRACHELBEN 7.7
5 HSCULLYLEE 21.29 SSH
EPARDCLARA 3000 HTIBBS
DANIEL 7.25

READY.
```

Figure G.3: GET#—Sample Output

Finally, line 50 sends control back up to the GET# statement to read the next character of the file:

```
50 GOTO 20
```

GOSUB (command word)

The GOSUB statement sends control of a program to a *subroutine*. A subroutine is a sequence of program statements, grouped together to perform a certain task. GOSUB instructs the computer to interrupt the usual line-by-line sequential order of program execution, go to the specified line of a subroutine, and begin executing the statements of that subroutine. Implied in the GOSUB command is that somewhere in the subroutine will be a RETURN statement. When the computer encounters the RETURN command, it returns control of the program to the statement immediately following the original GOSUB command and resumes the sequential execution of the program.

GOSUB may be used as either an *immediate command* or a program statement.

The GOSUB statement takes the following form:

```
GOSUB L
```

where L is the number of the first line of the subroutine. L must be expressed as a literal numeric value; for example:

```
GOSUB 600
```

The computer's response to GOSUB is always the same: control of the program moves to the indicated line number and proceeds sequentially until a RETURN command is encountered.

In many programming situations you may want to choose among several different subroutines, depending on the value of a certain variable. For such situations, the Commodore computers' version of BASIC offers the ON...GOSUB statement. For example, given a value of S from 1 to 5, the following statement will send control to one of five subroutines:

```
ON S GOSUB 100,200,300,400,500
```

In the ON...GOSUB statement, BASIC evaluates the expression after ON, finds its integral value, and then uses that value to decide which of a list of subroutines to call. If the value is 1, it calls the first subroutine in the list; if the value is 2, it calls the second; and so on. You will be able to study the action of the ON...GOSUB statement in the sample program below. (*Also see the entry under* ON for more information.)

From the programmer's point of view, there are several good reasons for taking the trouble to isolate certain programming tasks into individual subroutines. First, in many programs there are tasks that need to be performed more than once at different points during the performance. It would be a waste of both computer memory space and programming time to repeat the actual instruction lines for these tasks at several different points in the program listing, so the obvious solution is to write these instructions once, isolate them in a subroutine, and "call" the subroutine via a GOSUB statement whenever the task must be performed.

Another, perhaps even more essential, reason for using subroutines is to help achieve a well-organized, "modular" program structure. Experienced programmers have long realized, often to their chagrin, that a long and complex computer program can begin at a certain point to take on a life of its own and become terribly difficult to control, correct, or revise. One way to combat this phenomenon is to organize a long program into short and hopefully controllable tasks, which interact sequentially or collectively to accomplish the whole job that the program is written to perform. In this technique, each individual task is assigned to its own subroutine and the top, controlling part of the program—sometimes called the "main program" section—becomes, essentially, a series of subroutine calls, or GOSUB statements. Some programming languages—Pascal, for example—are specifically designed for this variety of modular, top-down programming. While BASIC is lacking in a few of the essential characteristics that make modular programming most successful, a BASIC program can often be vastly improved if organized into small, tidy, easily understood and easily revised subroutines.

Finally, the more programs you write, the more often you will find yourself writing instructions for similar—or, indeed, identical—tasks, time after time, for program after program. If you get into the habit of creating short subroutines for such commonly required tasks, you will be able to "transport" many of these subroutines verbatim, or with minimal revision, to functional places in other programs, thus streamlining the job of creating new computer programs.

Sample Program

Figure G.4 shows what is actually just the skeleton of a *menu*-driven program organized in a modular, top-down structure. The program doesn't actually do anything useful, but will serve all the same to illustrate, in the abstract, some of the principles of good program structure. It may also serve as a template for menu-driven programs that you may want to write for real jobs.

Lines 1 to 240 form the "main program" section, which controls the action of the program, and has three main jobs to perform:

1. display the menu on the screen;
2. accept and validate the user's menu choice; and
3. call the appropriate subroutines to implement the user's menu choice.

A menu, in the context of computer software, is simply a message to the user displaying the options available at any given point in a program performance. In addition to showing the options, a menu should indicate a clear and simple way for the user to express a choice among these options. This program's menu, produced by lines 10 to 120 of the program, is displayed in Figure G.5. There are four options, labeled ONE, TWO, THREE, and QUIT. To activate one of these options, the user need only type and enter a single digit—1, 2, 3, or 4. (The fourth option allows the user to terminate the program run.)

You can probably think of many situations where a menu similar to this one would be an appropriate way to offer choices to the user. For example, the program might be designed to play some sort of video game, and the menu could offer the user three levels of difficulty:

```
1) EASY GAME
2) MODERATELY DIFFICULT GAME
3) VERY DIFFICULT GAME
4) QUIT
```

```
  1 PRINT CHR$(147)
  3 REM
  5 REM **************
  7 REM *MAIN PROGRAM*
  8 REM **************
  9 REM
 10 PRINT:PRINT:PRINT
 15 LET M$=""
 20 PRINT "   MENU"
 30 PRINT "   ----"
 40 PRINT
 50 PRINT "   1) OPTION ONE"
 60 PRINT
 70 PRINT "   2) OPTION TWO"
 80 PRINT
 90 PRINT "   3) OPTION THREE"
100 PRINT
110 PRINT "   4) QUIT"
120 PRINT : PRINT
130 GOTO 160
140 PRINT CHR$(145);
150 PRINT "   1,2,3, OR 4 ";
160 PRINT TAB(15);
170 INPUT M$
180 IF M$<"1" OR M$>"4" OR LEN(M$)>1 GOTO 140
190 PRINT CHR$(147)
200 GOSUB 800
210 ON VAL(M$) GOSUB 300,400,500,600
```

Figure G.4: GOSUB—Sample Program

```
220 GOSUB 700
230 PRINT CHR$(147)
240 GOTO 10
300 REM *OPTION ONE*
310 PRINT "     OPTION ONE"
320 RETURN
400 REM *OPTION TWO*
410 PRINT "     OPTION TWO"
420 RETURN
500 REM *OPTION THREE*
510 PRINT "    OPTION THREE"
520 RETURN
600 REM * HALT *
610 PRINT "  END OF PROGRAM."
620 GOSUB 800
630 END
700 REM *CONTINUE*
705 GOSUB 800
710 FOR I = 1 TO 10 : PRINT : NEXT I
720 INPUT "   CONTINUE";A$
730 RETURN
800 REM * LINE *
805 PRINT
810 FOR I = 1 TO 21
820   PRINT "*";
830 NEXT I
840 PRINT : PRINT
850 RETURN
```

Figure G.4: GOSUB—Sample Program, continued

Or the program might be preparing a financial report, and the menu could offer a choice of different reporting methods for one aspect of the report:

```
1) STRAIGHT-LINE DEPRECIATION
2) SUM-OF-THE-YEARS' DIGITS DEPRECIATION
3) DOUBLE-DECLINING-BALANCE DEPRECIATION
4) QUIT
```

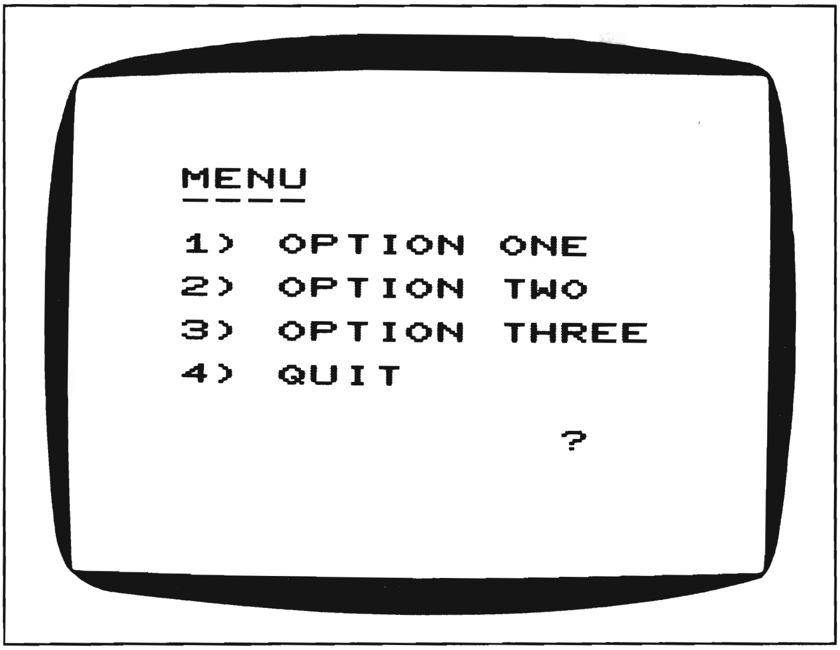
Or finally, the program could have a graphics capability, and offer the user a choice of one variable to graph as a function of another:

```
1) GRAPH Y = F(X)
2) GRAPH X = F(Z)
3) GRAPH Z = F(Y)
4) QUIT
```

Whatever the options of the program might be, the purpose of the menu is to present them clearly and to elicit an unambiguous response from the user expressing a choice among them. Line 170 of the program reads a character from the keyboard, and line 180 tests to make sure that the character is within the appropriate range of menu choices:

```
170 INPUT M$
180 IF M$< "1" OR M$> "4" OR LEN(M$)> 1 GOTO 140
```

The response is read as a string (M$), rather than as a numeric value, in order to allow for input errors. If the user, by mistake, enters any inappropriate response (i.e., anything other than a single digit from 1 to 4), line 100 sends control of the program back to line 140, and a new message is placed on



```
MENU
----
1)  OPTION  ONE
2)  OPTION  TWO
3)  OPTION  THREE
4)  QUIT

                  ?
```

Figure G.5: GOSUB—Sample Output, Menu

the screen to prompt the user to try again. Figure G.6 shows this message.

Once the computer has elicited an appropriate menu choice, the program continues on to a series of subroutine calls, in lines 200 to 220. The subroutines that the program can call are located at lines 300, 400, 500, 600, 700, and 800.

The first call is to the subroutine at line 800:

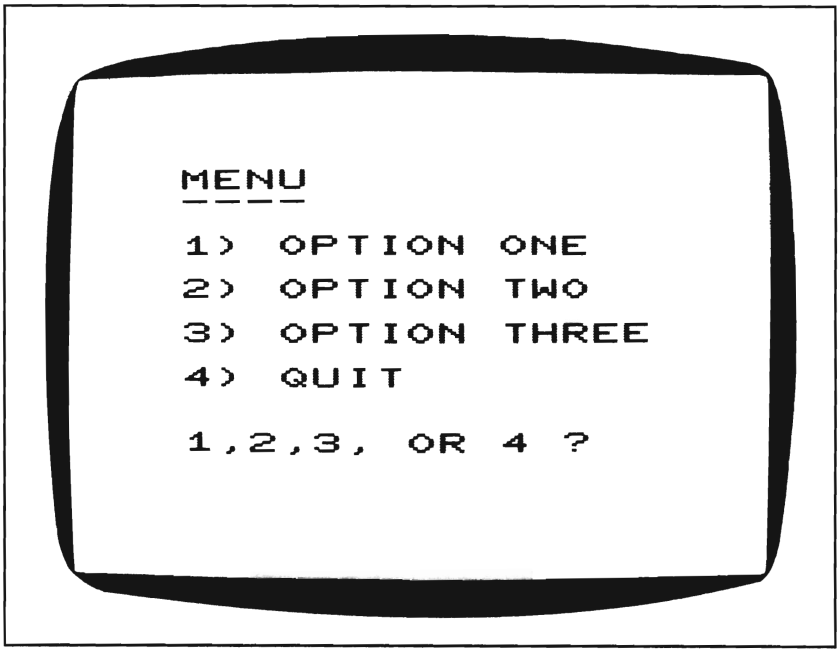
```
200 GOSUB 800
```

This subroutine simply prints a row of asterisks across the screen. As simple as it is, this subroutine is representative of many important subroutines you might write just to arrange some visual detail of a screen display. This subroutine is called from two other places in the program.

The next GOSUB statement, at line 210, calls one of the four "option" subroutines. The statement uses the ON...GOSUB syntax to choose among the list of subroutines:

```
210 ON VAL(M$) GOSUB 300, 400, 500, 600
```

Since M$ contains a value from "1" to "4", the expression VAL(M$)



```
MENU
----
1)  OPTION  ONE
2)  OPTION  TWO
3)  OPTION  THREE
4)  QUIT

1,2,3,  OR  4  ?
```

Figure G.6: GOSUB—Sample Output, Menu with Input Error Message

returns an integer from 1 to 4, which in turn points to one of the subroutines in the list, from the first to the fourth.

You'll notice that the option subroutines in this program are really nothing but "stubs" of subroutines. All they do is display an identifying message on the screen. You may often find yourself writing such subroutines while you plan the overall structure of your program. When you are ready, you can go back and fill in the details of the subroutines themselves.

The final GOSUB statement calls a very important subroutine:

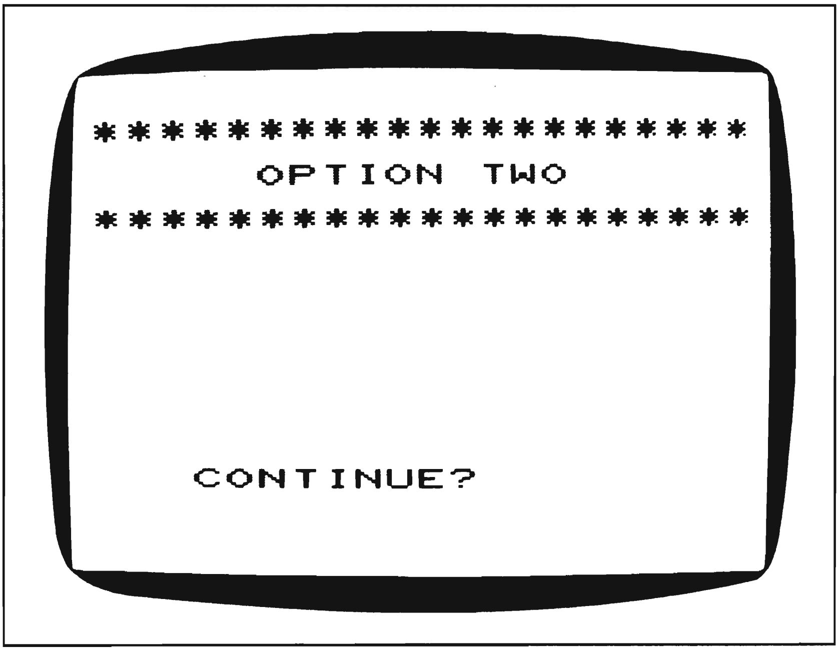
```
220 GOSUB 700
```

The subroutine at line 700 allows the user to examine a screenful of information at leisure before moving on to the next activity of the program. The subroutine prints the query:

```
CONTINUE?
```

on the screen, and then waits for the user to enter some response at the keyboard:

```
720 INPUT "CONTINUE"; A$
```



```
*********************
     OPTION TWO
*********************






   CONTINUE?
```

Figure G.7: GOSUB—Sample Output, "Continue"

Figure G.7 shows the result of this subroutine. If the option subroutine had filled the screen with information, the user would be able to study the screen for as long as necessary, and then simply press the RETURN key to continue the program run.

Notes and Comments

— It's always a good idea to identify subroutines with REM lines, as in Figure G.4. You should think of an appropriate descriptive title for each subroutine you write.

— A program will terminate with an *error message* if a GOSUB statement attempts to send control to a nonexistent line number. To demonstrate this error, run the following two-line program:

```
10 GOSUB 20
30 END
```

The computer will display the following error message on the screen:

```
? UNDEF'D STATEMENT
  ERROR IN 10
```

— Likewise, a program will terminate with an error message if a RETURN statement is encountered without the direction of a GOSUB statement. Run the following one-line program to see the results of this error:

```
10 RETURN
```

The computer displays the message:

```
? RETURN WITHOUT GOSUB
ERROR IN 10
```

GOTO (command word)

The GOTO statement sends control of the current program to a specified line. The line number must be expressed as a *literal value*; for example:

```
GOTO 10
```

GOTO may be used as an *immediate command* or as a program instruction. If GOTO is part of a program, the direction of the "jump" caused by the

GOTO statement may be up to an earlier line in the program:

```
90 GOTO 10
```

or down to a later point in the program:

```
100 GOTO 150
```

or even back to the beginning of the very same line:

```
20 GET A$ : IF A$ = "" GOTO 20
```

This line illustrates the use of GOTO in an IF statement. In such a conditional GOTO instruction, the jump will only be performed if the *logical expression* contained in the IF statement (A = "" in this case) is evaluated as true.

Sample Program

The program shown in Figure G.8 will help you to balance your checkbook by giving you a record of withdrawals, and the balance after each withdrawal. The program begins by asking you for the previous balance forward (from the last time you balanced your checkbook), and for the

```
 10 DEF FN R(X) = INT(100*X+.5)/100
 15 PRINT CHR$(147)
 20 INPUT "BALANCE FORWARD";B
 30 INPUT "# OF LAST CHECK";N
 35 PRINT CHR$(147)
 40 GOSUB 100
 45 PRINT TAB(12);B
 50 GOSUB 200
 55 IF B<0 GOTO 85
 60 GOSUB 300
 65 PRINT N;TAB(4);A;TAB(12);FN R(B)
 70 LET V = V + 1
 75 IF V>18 GOTO 35
 80 GOTO 50
 85 GOSUB 300
 90 PRINT N;TAB(4);A;" OVERDRAFT"
 95 END
100 PRINT " #   AM'T    BALANCE"
105 PRINT
```

Figure G.8: GOTO—Sample Program

```
110 LET V=3
115 RETURN
200 GOSUB 400
205 PRINT "                              "
210 GOSUB 400
220 LET N = N + 1
230 PRINT "CHECK #";N;
240 INPUT A
250 LET B = B - A
260 RETURN
300 PRINT CHR$(19)
310 FOR I = 1 TO V
320   PRINT
330 NEXT I
340 RETURN
400 PRINT CHR$(19)
410 FOR I = 1 TO 20
420   PRINT
430 NEXT I
440 RETURN
```

Figure G.8: GOTO—Sample Program, continued

number printed on the last check that you reconciled against your account. These two values are assigned to the variables B and N, respectively.

The main action of the program is controlled by lines 35 to 95. Line 40 calls the subroutine at line 100 to print the column headings. This subroutine also initializes the value of the variable V, which is a counter for the number of lines of information displayed on the screen at any given point in the program run. Line 60 calls the subroutine at line 200 for the actual input of a check amount. This subroutine increments the check number, N (line 220); prints an input prompt at the bottom of the screen (line 230); reads the check amount (line 240); and finally finds the new balance, B (line 250).

Back in the "main program" section, lines 55 to 80 contain a series of three GOTO statements (two of them conditional GOTOs) that determine what will happen next in the program. First, as soon as control of the program returns from the subroutine at line 200, line 55 tests the value of B, the checkbook balance:

```
55 IF B<0 GOTO 85
```

The purpose of this test is to provide contingency measures in the event of an overdraft. Paraphrased, the statement says, "If the last check decreased the checkbook balance to an amount less than zero, go to line 85." The instruction in line 90 writes an overdraft message on the screen, and then line 95 terminates the program.

If there is no overdraft, control of the program continues down to lines 60 and 65, which print the check number, the amount, and the new balance at line V on the screen:

```
60 GOSUB 300
65 PRINT N; TAB(4); A; TAB(12); FN R(B)
```

(The subroutine at line 300 simply positions the cursor at line V. The user-defined function R, defined in line 10 of the program, rounds the balance B to the nearest cent. *See* DEF FN and FN for an explanation of user-defined functions.)

Line 70 increments the value of V and then line 75 tests to see if the screen is full:

```
70 LET V = V + 1
75 IF V > 18 GOTO 35
```

If V is greater than 18 (meaning the screen is full, except for the lines at the bottom reserved for the input prompt), control is sent back up to line 35. Lines 35 to 45 clear the screen, print the column headings again, and display the current checkbook balance in the right-hand column:

```
35 PRINT CHR$(147)
40 GOSUB 100
45 PRINT TAB(12); B
```

(Note that the Commodore ASCII code character 147 clears the screen. *See the entry under* CHR$ for details.)

Finally, if neither of the conditional GOTOs—line 55, the overdraft test, or line 75, the full-screen test—has changed the course of the program run, control drops down to line 80:

```
80 GOTO 50
```

This statement simply sends control of the program back up to the beginning of the block of instructions that process each check, thus forming a loop that allows you to enter as many checks as you want.

A sample run of the program appears in Figure G.9.

Notes and Comments

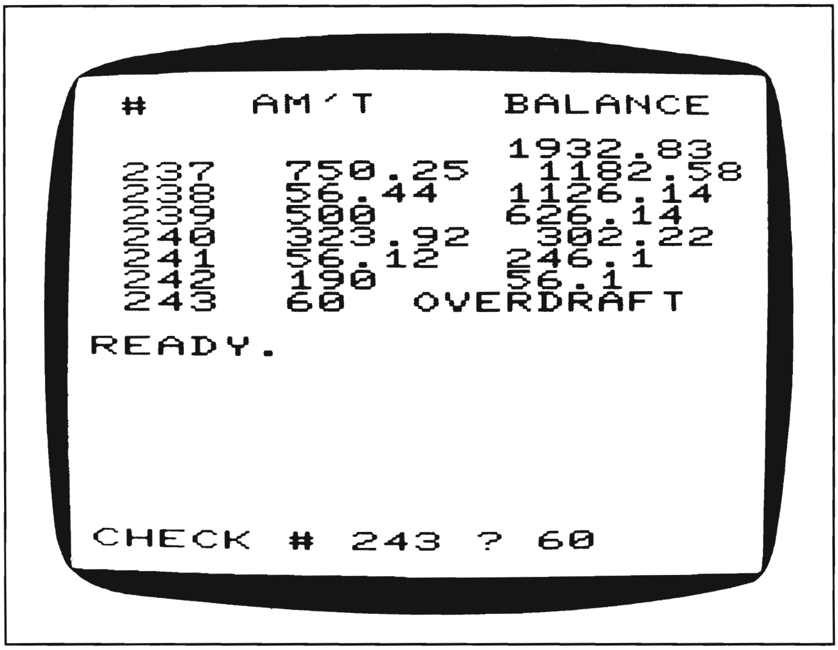
— You can use the GOTO statement as an immediate command to instruct the computer to begin the performance of a program. For example, if the current program in the computer's memory begins at line 10, you can enter the command:

```
GOTO 10
```

to start the program. The advantage of using GOTO in this way is that you will not lose any variables left over from previous runs of the program. (The RUN command clears all variables before starting execution of the program.)

— A conditional GOTO statement may be written in any of three different formats. For example, all three of the following statements mean the same thing:

```
IF B<0 THEN 90
IF B<0 THEN GOTO 90
IF B<0 GOTO 90
```



```
#      AM'T        BALANCE
                   1932.83
237    750.25       1182.58
238    56.44       1126.14
239    500         626.14
240    323.92       302.22
241    56.12       246.1
242    190         56.1
243    60     OVERDRAFT
READY.



CHECK # 243 ? 60
```

Figure G.9: GOTO—Sample Output

— BASIC provides the ON...GOTO statement to choose among a list of line numbers for the transfer of control; for example:

```
ON V GOTO 100,200,300,400
```

If V represents a value from 1 to 4 in this statement, the first, second, third, or fourth line number in the list will be chosen. (*See the entries under* ON and GOSUB for more details.)

IF (command word)

The IF statement allows you to incorporate a decision-making capacity into a BASIC program. The syntax of an IF statement makes use of another BASIC word, THEN. The general form of an IF statement is:

IF (logical expression) **THEN** (command)

When the computer performs an IF statement, it evaluates the *logical expression* to either true or false. If the expression is true, then the computer performs the command that is stated after THEN. If the logical expression is false, then the IF statement will result in no action, and the computer will simply continue on with the program.

Logical expressions are statements of equality or inequality that are either true or false. You can write such expressions using one or more of the following symbols:

=	("is equal to")
< >	("is not equal to")
<	("is less than")
>	("is greater than")
<=	("is less than or equal to")
>=	("is greater than or equal to")

The BASIC words AND, OR, and NOT can also be used to build or modify logical expressions. (*See* the entries under these words for more information.)

The action after the word THEN in an IF statement can be expressed as any BASIC command word.

Here are three examples of IF statements, followed by paraphrases of what they do:

```
IF HOUR > 12 THEN LET HOUR = HOUR - 12
```

"If the variable HOUR contains a value that is greater than 12, then store a new value in HOUR, equal to 12 less than the previous value."

```
IF AGE = 65 THEN GOSUB 300
```

"If the variable AGE contains the value 65, then perform the subroutine located at line 300."

```
IF T <= N THEN INPUT N
```

"If the value of T is less than or equal to the value of N, then read a new value for N from the keyboard."

The logical expression in an IF statement can also compare two strings, as in the following example:

```
40 IF A$(I) < A$(J) THEN GOTO 80
```

This statement results in a character-by-character comparison of the ASCII character codes in each string; it could be part of a program that alphabetizes (or "sorts") the strings in the array A$. (Such a program is listed and described under the heading *Algorithm.*)

Sample Program

The program shown in Figure I.1 is a version of a classic computer guessing game called "over/under." In this version of the game, the computer chooses, at random, a number between 1 and 100 and gives you seven chances to guess the right number. After each guess the computer tells you if your guess is "over" or "under" the correct number.

At the heart of this program is a series of IF statements that enable the computer to evaluate your guess and to make the decisions that control the game.

Lines 10 to 80 of the program display a set of instructions at the top of the screen. Line 90 uses the RND function in a formula that ensures that the computer's number will be between 1 and 100, inclusive. The number is assigned to the variable N. Line 100 sets up a counter, I, to count the number of tries you have taken. Finally, line 120 reads your guess from the keyboard, and assigns it to the variable G, so the program can begin comparing your guess to the correct number.

The first decision statement tests to see if you have guessed correctly:

```
140 IF G = N GOTO 230
```

If so, then control of the program goes to line 230, which tells you that you have guessed the right answer.

The next two IF statements print either OVER or UNDER on the screen if your guess is not correct:

```
150 IF G < N THEN PRINT "UNDER"
160 IF G > N THEN PRINT "OVER"
```

```
 10 PRINT CHR$(147)
 20 PRINT "OVER/UNDER"
 30 PRINT
 40 PRINT "I AM THINKING OF A"
 50 PRINT "NUMBER FROM 1 TO 100."
 60 PRINT "YOU MAY HAVE 7 TURNS"
 70 PRINT "TO GUESS IT."
 80 PRINT
 90 LET N = 1 + INT(RND(0)*100)
100 LET I = 1
110 PRINT I;
120 INPUT G
130 PRINT CHR$(145);TAB(8);"==> ";
140 IF G=N GOTO 230
150 IF G<N THEN PRINT "UNDER"
160 IF G>N THEN PRINT "OVER"
170 LET I = I + 1
180 IF I<=7 GOTO 110
185 PRINT
190 PRINT "SORRY. THE NUMBER"
200 PRINT "WAS";N;"."
210 PRINT
220 GOTO 250
230 PRINT "RIGHT!"
240 PRINT
250 INPUT "ANOTHER GAME"; A$
260 IF LEFT$(A$,1)="Y" GOTO 10
```

Figure I.1: IF—Sample Program

Next, the counter I is incremented by 1, and a final IF statement looks to see if you have used up all your chances:

```
180 IF I <= 7 GOTO 110
```

If you still have turns left, then control of the program jumps up to line 110 to start the process over again. Otherwise, if the expression:

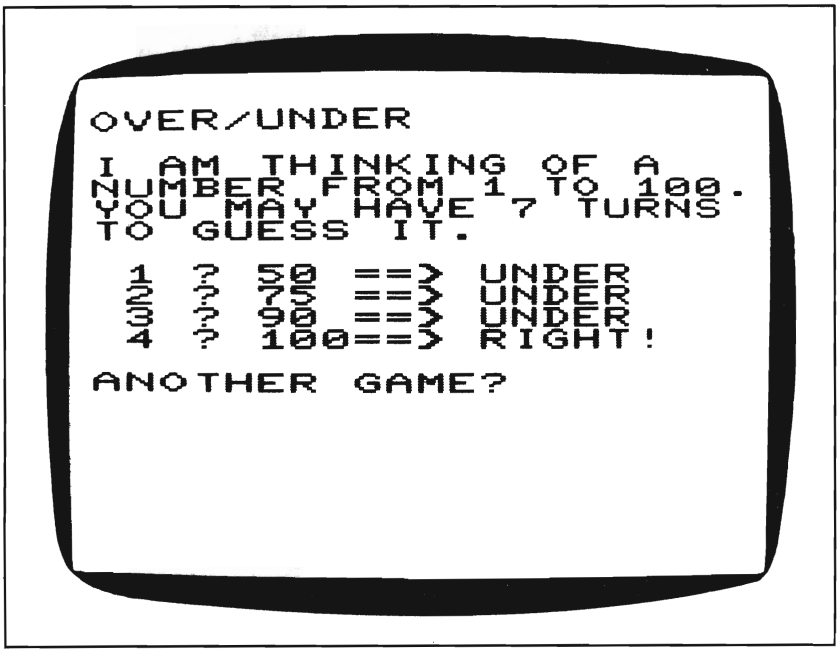
```
I <= 7
```

is false, then line 180 results in no action, and the program prints the regretful message of lines 190 and 200. Lines 250 and 260 offer you another round when the game is over.

Figure I.2 shows a sample game.

Notes and Comments

— Sometimes it can be to your advantage to know how BASIC evaluates logical expressions. The result of this evaluation is



```
OVER/UNDER

I AM THINKING OF A
NUMBER FROM 1 TO 100.
YOU MAY HAVE 7 TURNS
TO GUESS IT.

  1 ? 50 ==> UNDER
  2 ? 75 ==> UNDER
  3 ? 90 ==> UNDER
  4 ? 100==> RIGHT!

ANOTHER GAME?
```

Figure I.2: IF—Sample Output

actually coded numerically, as follows:

false = 0
true = -1

This means that you can, if you want to, replace the logical expression in an IF statement with a simple numeric variable; for example:

```
IF N THEN PRINT "HELLO"
```

If the value of N is 0 in this statement, the computer will react as though you had actually put a logical expression in the IF statement, and the expression was false. If N has any value other than 0 (not just -1), the computer will read it as a "true" logical expression, and the PRINT command at the end of the IF statement will be performed.

— BASIC allows more than one form for a conditional GOTO (i.e., an IF statement in which the command after THEN is GOTO). For details, *see* the "Notes and Comments" section under the heading GOTO.

— Since BASIC allows a program to contain more than one statement (with the multiple statements separated by colons), the following question arises: Will any statements located on the same line as an IF statement be executed if the logical expression in the IF statement is false? In other words, given the following general form:

```
IF (logical expression) THEN (statement#1) : (statement#2)
```

we know that if the logical expression is false, statement#1 will not be performed; but what about statement#2?

Unfortunately, the answer to this question varies from one version of BASIC to the next; this can cause serious and extremely elusive "bugs" in BASIC programs that are translated from one version to another. For Commodore BASIC, the answer is as follows: If the logical expression is false, the computer moves on to the *next line* of the program, without performing *any* of the commands located on the same line as the IF statement itself. Consider an example:

```
IF I = J THEN PRINT "ONE" : PRINT "TWO"
```

In this statement, if the expression I = J is true, the output display resulting from the command will be as follows:

```
ONE
TWO
```

On the other hand, if the expression I = J is false, there will be no output display at all from this line. In other words, neither of the PRINT statements after THEN will be performed.

Immediate Command (computer vocabulary)

An immediate command (also referred to as a "direct" command) is one that the computer performs as soon as you enter it from the keyboard, as opposed to one that is part of a program. Unlike the lines of a BASIC program, immediate commands are not numbered. Many BASIC commands may be used either as immediate commands or as program statements.

INPUT (command word)

The INPUT command instructs the computer to wait for data to be typed from the keyboard. When the data is entered, the computer assigns it to the *variable* named in the INPUT statement. The simplest form of the INPUT instruction is:

```
INPUT V
```

where V is any variable name. The variable type can be real, integer, or *string*; the computer, in turn, expects the input value entered from the keyboard to correspond in type to the variable in the INPUT statement. INPUT may not be used as an *immediate command.*

When performing an INPUT statement, the computer displays a question mark (?) on the screen to indicate that it is waiting for input data. (The sample program below will help you explore this feature.) Each character of input data is "echoed" on the screen as you type it from the keyboard. Pressing the RETURN key completes the data input.

In addition, BASIC allows you to include a prompt string in the INPUT statement. When the statement is performed, the computer displays your prompt, followed by a question mark, and then waits for the appropriate input data. With the prompt string, the INPUT statement takes the following form:

```
INPUT "PROMPT STRING"; V
```

Finally, INPUT allows more than one data element to be read by a single statement; for example:

```
INPUT "THREE NUMBERS"; V1, V2, V3
```

This statement will display the following prompt on the screen:

```
THREE NUMBERS?
```

and then wait for three numerical data items to be entered from the keyboard. The data items may all be typed on one line, with commas separating each number:

```
THREE NUMBERS? 21, 37, 52
```

or each number can be entered on a line of its own, followed by RETURN. In this case, the computer will display *two* question marks for each data item after the first:

```
THREE NUMBERS? 21
?? 37
?? 52
```

Sample Program

You can use the program shown in Figure I.3 to explore the computer's reaction to the various forms of the INPUT statement. The program consists simply of a series of five INPUT statements (lines 30, 60, 90, 120, and 150), representing the variety of forms the command can take. Before each INPUT statement, a PRINT line displays on the screen a brief message explaining the kind of INPUT statement that is coming up. Figure I.4 shows a sample run of the program.

Notes and Comments

— If you wish to enter a string value that contains a comma, the entire string must be enclosed in quotation marks; for example, in response to the following INPUT statement:

```
INPUT S$
```

you may enter:

```
? "JONES, D."
```

In this case, the string variable S$ will be assigned a string that is nine characters long:

```
JONES, D.
```

If you fail to enclose such a string in quotation marks, the computer will read it as two different strings, separated by a comma. Since, in this case, the INPUT statement contains only one variable that is to be assigned a value, the computer will display the following *error message:*

```
?EXTRA IGNORED
```

This "extra" refers to the second string input, located after the comma.

— If you enter a string value when the computer is expecting numerical input, an error message will appear on the screen, prompting you to reenter the data. The message is:

```
?REDO FROM START
?
```

The second question mark is your prompt to reenter the data. After such an input error, BASIC repeats the performance of the *entire* INPUT statement; if the statement contains more than one variable, you will have to reenter all the data values.

```
 10 PRINT CHR$(147)
 20 PRINT "NUMERICAL INPUT:"
 30 INPUT V
 40 PRINT
 50 PRINT "STRING INPUT:"
 60 INPUT V$
 70 PRINT
 80 PRINT "THREE NUMBERS:"
 90 INPUT V1,V2,V3
100 PRINT
110 PRINT "INPUT PROMPT:"
120 INPUT "STRING, INTEGER";S$,I%
130 PRINT
140 PRINT "INPUT PROMPT:"
150 INPUT "INTEGER, STRING";I%, S$
```

Figure I.3: INPUT—Sample Program

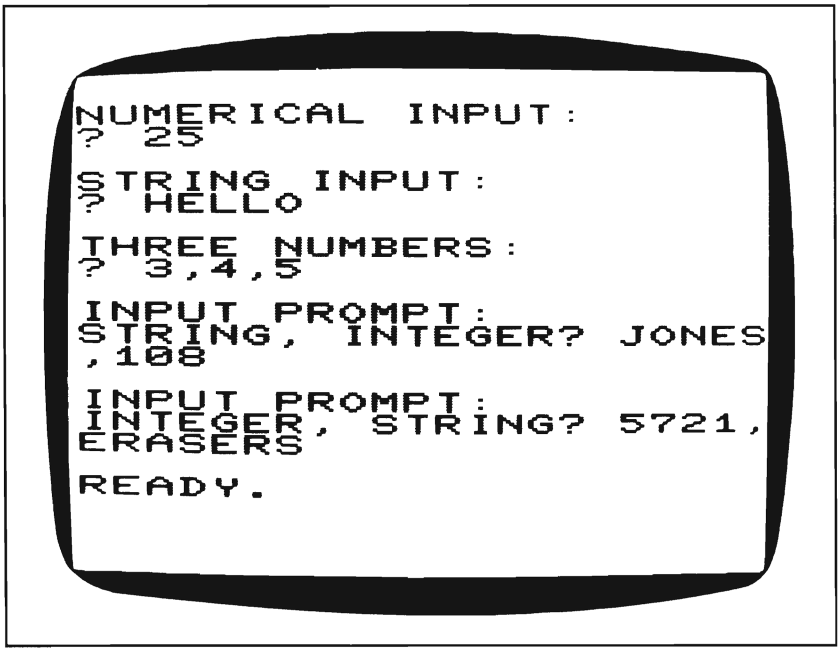
— If, in response to an INPUT statement, you press the RETURN key without typing any data, the variable named in the INPUT statement will keep whatever value it had before the statement was performed. To demonstrate this characteristic, run the following short program:

```
10 LET I = 27
20 LET S$ = "COMPUTER"
30 INPUT I
40 INPUT S$
50 PRINT I, S$
```

In response to the two input prompts, press only the RETURN key. After both INPUT statements have been performed, the program will print the following line on the screen:

```
27          COMPUTER
```

This shows that both variables, I and S$, still contain the same values they held before the INPUT statements.



```
NUMERICAL  INPUT:
? 25

STRING  INPUT:
? HELLO

THREE  NUMBERS:
? 3,4,5

INPUT  PROMPT:
STRING,  INTEGER? JONES
,108

INPUT  PROMPT:
INTEGER,  STRING? 5721,
ERASERS

READY.
```

Figure I.4: INPUT—Sample Output

INPUT# (data input command word)

The INPUT# command reads data items from an external data *file* (stored on tape or disk), and assigns those items to *variables*. The format of INPUT# is as follows:

```
INPUT#F, V
```

where F is the file number, and V is the variable that the data item will be stored in. INPUT# also allows a list of variables, in which case a data item will be read for each variable in the list:

```
INPUT#F, V1, V2, V3, ...
```

Unlike the GET# command, which reads external files one character at a time, INPUT# reads entire data items at a time. INPUT# recognizes the RETURN character and the comma as separators between data items in a file. When you create a data file, you must make sure that one of these characters is printed between each item of the file. (*See the entry under* PRINT#.) Probably the most commonly used data item separator is the RETURN character (ASCII code 13).

INPUT# must be preceded by an OPEN statement that identifies the file and its location. INPUT# may not be used as an immediate command.

Sample Program

The program listed in Figure I.5 is designed to read the data file named EMPLOYEES and to produce a table from that file. EMPLOYEES is created by the sample program described under the heading PRINT#; it is a disk file that contains information about several employees in an imaginary company. There are four data items for each employee:

1. a "tag" that indicates whether the employee is on an hourly wage ("H") or a monthly salary ("S");
2. the employee's last name;
3. the employee's first name;
4. the employee's salary: hourly if the tag is H; monthly if the tag is S.

The very first data item in the file is a number indicating how many employee records are stored in the file.

The INPUT# program begins by opening the file for reading, in line 10:

```
10 OPEN 1,8,2, "EMPLOYEES,S,R"
```

Notice that the file number is 1; each INPUT# command in the program must therefore refer to this number. Line 20 reads the first data item of the file into the variable N:

20 INPUT#1, N

Since N represents the number of employees described in the file, the program can dimension the four arrays that will ultimately store the data—T$ for the "tags"; L$ for the last names; F$ for the first names; and S for the salaries:

30 DIM T$(N), L$(N), F$(N), S(N)

```
 10 OPEN 1,8,2,"EMPLOYEES,S,R"
 20 INPUT#1, N
 30 DIM T$(N),L$(N),F$(N),S(N)
 40 FOR I=1 TO N
 50 INPUT#1,T$(I),L$(I),F$(I),S(I)
 60 NEXT I
 65 CLOSE 1
 70 PRINT CHR$(147)
 80 PRINT "    EMPLOYEE LIST"
 90 PRINT : PRINT
100 PRINT "SALARIED:"
105 PRINT
110 FOR I = 1 TO N
120 IF T$(I) = "H" GOTO 140
130 GOSUB 200
140 NEXT I
143 PRINT : PRINT
145 PRINT "HOURLY:"
147 PRINT
150 FOR I = 1 TO N
160 IF T$(I) = "S" GOTO 180
170 GOSUB 200
180 NEXT I
190 PRINT:PRINT:END
200 PRINT L$(I);","; LEFT$(F$(I),1);"."; TAB(14);S(I)
210 RETURN
```

Figure I.5: INPUT#—Sample Program

Finally, the rest of the file is read from within a FOR loop, four data items at a time. Each four items represent one complete employee record:

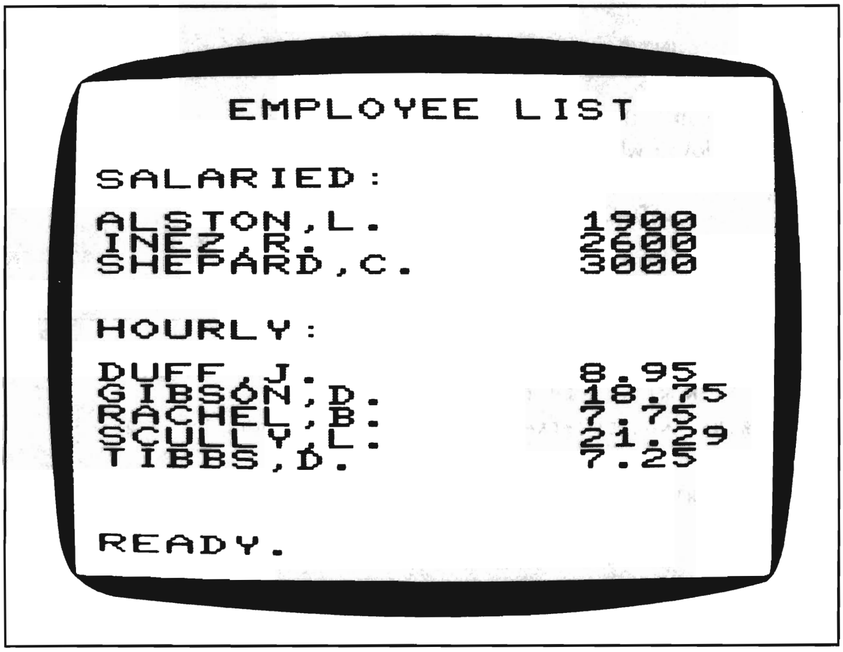
```
40 FOR I = 1 TO N
50 INPUT#1, T$(I), L$(I), F$(I), S(I)
60 NEXT I
```

Notice that the control variable I is used to increment the indexes into the arrays, so that the four data items of each record are stored in corresponding elements of the four arrays.

When all the data has been read, the file can be closed:

```
65 CLOSE 1
```

The rest of the program (lines 70 to 210) produces a display of the employee data, under two headings—SALARIED and HOURLY. Figure I.6 shows the output from the program. Notice in lines 120 and 160 that the values stored in the array T$, the tags, are used to decide the category of each employee.



```
        EMPLOYEE LIST

SALARIED:
ALSTON,L.                1900
INEZ,R.                  2600
SHEPARD,C.               3000

HOURLY:
DUFF,J.                  8.95
GIBSON,D.                18.75
RACHEL,B.                7.75
SCULLY,L.                21.29
TIBBS,D.                 7.25


READY.
```

Figure I.6: INPUT#—Sample Output

Notes and Comments

— INPUT# can also read a data file stored on a cassette tape. If EMPLOYEES were stored on cassette, line 10, the OPEN command, would have to be revised as follows:

```
10 OPEN 1,1,0, "EMPLOYEES"
```

The program would need no other changes to read a cassette file.

INT (function)

The INT *function* supplies the integral value of a number. In the case of positive numbers, INT simply eliminates the fractional portion of the number. For example, the expression:

INT(2.7)

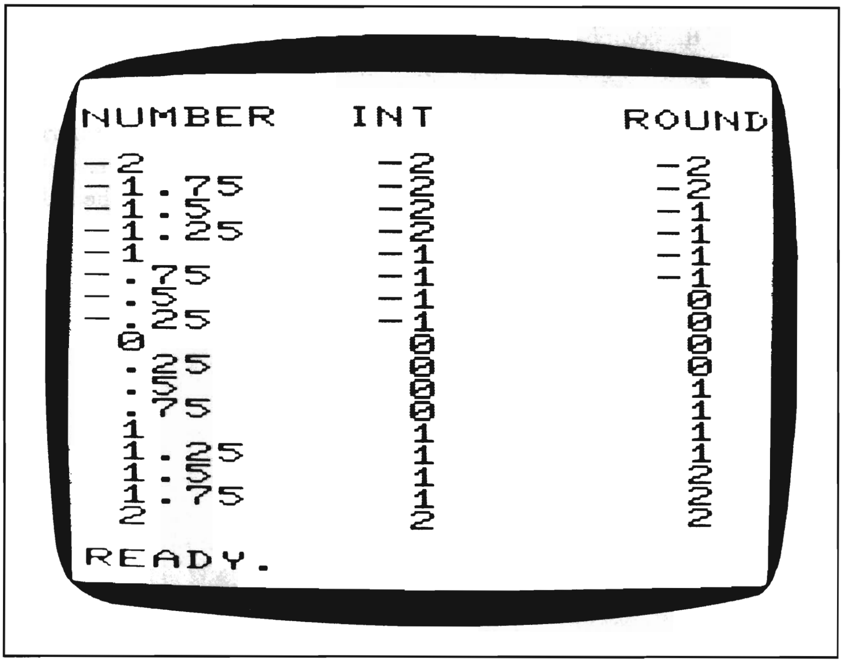
would result in the value 2. In the case of negative numbers, INT supplies the next lower whole number. For example:

INT(−2.7)

would return the value −3.

```
 5 PRINT CHR$(147)
10 DEF FN R(X) = INT(X+.5)
20 PRINT "NUMBER";TAB(8);"INT";TAB(17);"ROUND"
30 FOR N = -2 TO 2 STEP .25
40   PRINT N;TAB(9);INT(N);TAB(18);FN R(N)
50 NEXT N
```

Figure I.7: INT—Sample Program



```
NUMBER      INT          ROUND
-2          -2           -2
-1.75       -2           -2
-1.5        -2           -1
-1.25       -2           -1
-1          -1           -1
-.75        -1           -1
-.5         -1            0
-.25        -1            0
 0           0            0
 .25         0            0
 .5          0            1
 .75         0            1
 1           1            1
 1.25        1            1
 1.5         1            2
 1.75        1            2
 2           2            2

READY.
```

Figure I.8: INT—Sample Output

Sample Program

The program shown in Figure I.7 compares, for a series of *arguments,* the value returned by INT with the value returned by a common rounding formula. Line 10 of the program contains a user-defined function that rounds a number, X, to the nearest whole number:

```
10 DEF FN R(X) = INT(X + .5)
```

The argument, N, is the control variable of a FOR loop, and ranges in value from -2 to +2 in steps of .25 (line 30). Figure I.8 shows the output from the program.

Interactive (computer vocabulary)

The term *interactive* describes the computer's ability to respond, during the run of a program, to information that the user types at the keyboard. A program that takes advantage of this quality might create a "dialogue" between the computer and the computer user; in such a program, the course of the computer's action depends on the information that the user enters at the keyboard during the program's performance.

A good example of an interactive program dialogue can be found under the heading DIM. The program first tells the user exactly what information is needed, then prints a question-mark prompt for each item to be entered, and finally produces output (a bar graph) based on the values entered. Other examples can be found throughout the book.

See also Menu and User-Friendly.

LEFT$ (string function)

The LEFT$ *function* allows you to isolate the first *n* characters of a *string*. The function takes the form:

```
LEFT$(S$,N)
```

where S$ represents a string, and N an integer. S$ may be expressed as a literal string, a string *variable*, or a string expression (that is, a *concatenation*). The function returns the first N characters of S$.

Sample Program

The program in Figure L.1 demonstrates a *user-friendly* method of accepting a yes-or-no response from the keyboard. Many programming situations require such a response to determine the subsequent action of the program. The following are examples of questions that you can imagine appearing on the screen during the performance of various programs:

```
DO YOU WANT TO SEE ANOTHER REPORT?
ARE YOU READY TO CONTINUE?
DO YOU NEED HELP?
DO YOU WANT TO PLAY AGAIN?
DO YOU HAVE MORE DATA TO INPUT?
```

You can undoubtedly think of many more. Each of these questions requires the user to answer *yes* or *no* so that the computer can decide what to do next. In reading the answer, a program should be designed to accept a variety of valid answers, and yet to safeguard against the occasional invalid answer. Specifically, the following two points should be considered:

1. If you are answering affirmatively, you should be allowed to type "YES" or simply "Y". Likewise, for a negative answer, either "NO" or "N" should be acceptable.
2. If you make a typing error, the program should recognize it as such and give you another chance to type a valid answer. For example, if you should enter a "U" instead of a "Y", the program should recognize the answer as invalid.

The subroutine at line 100 (Figure L.1) is designed to meet these two requirements. It uses the LEFT$ function to determine whether your response is:

1. affirmative,
2. negative, or
3. invalid,

and it acts accordingly.

Line 110 places the following input prompt on the screen:

Y)ES OR N)O?

```
  5 PRINT CHR$(147)
 10 PRINT "CONTINUE?"
 20 GOSUB 100 : REM GET ANSWER
 30 IF F$="N" GOTO 80
 40 PRINT
 50 PRINT "CONTINUING PROGRAM..."
 60 PRINT
 70 GOTO 10
 80 PRINT "ENDING PROGRAM."
 90 END
100 REM ** YES OR NO
110 INPUT "Y)ES OR N)O";A$
120 LET F$ = LEFT$(A$,1)
130 IF NOT(F$ = "Y" OR F$ = "N") GOTO 150
140 RETURN
150 PRINT : PRINT "REENTER."
160 GOTO 110
```

Figure L.1: LEFT$—Sample Program

and reads an answer into the variable A$. Line 120, which illustrates the use of LEFT$, assigns the first character of the string A$ to the variable F$:

```
120 LET F$ = LEFT$(A$,1)
```

The next statement tests to see if the character stored in F$ represents a valid yes-or-no answer:

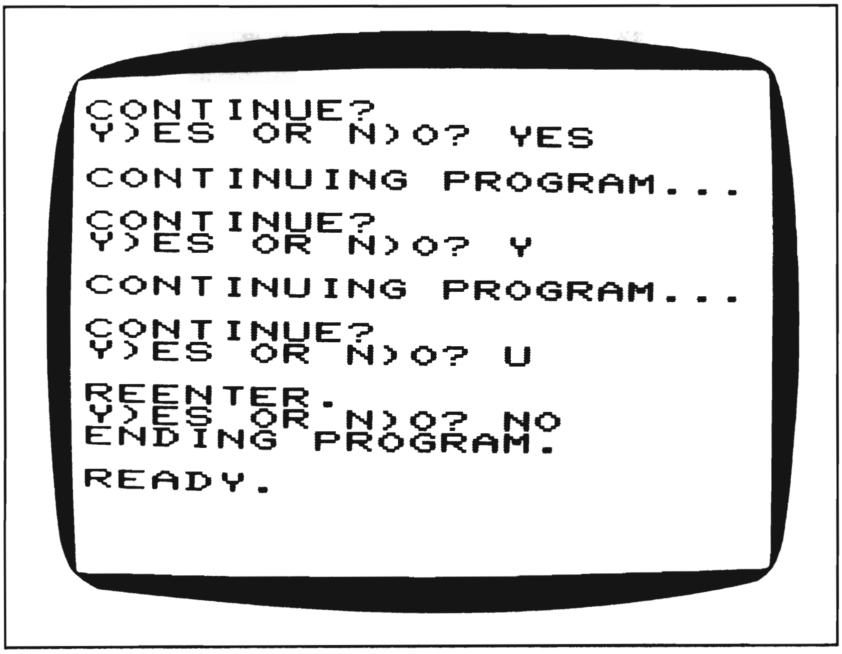
```
130 IF NOT(F$ = "Y" OR F$ = "N") GOTO 150
```

If F$ contains neither a "Y" nor an "N" character, control of the program goes down to line 150, which prints an error message; line 160 then loops back up to the INPUT statement:

```
150 PRINT : PRINT "REENTER."
160 GOTO 110
```

The result, of an invalid response, then, will be something like this:

```
Y)ES OR N)O? U

REENTER.
Y)ES OR N)O?
```



```
CONTINUE?
Y)ES OR N)O? YES

CONTINUING PROGRAM...

CONTINUE?
Y)ES OR N)O? Y

CONTINUING PROGRAM...

CONTINUE?
Y)ES OR N)O? U

REENTER.
Y)ES OR N)O? NO
ENDING PROGRAM.

READY.
```

Figure L.2: LEFT$—Sample Output

If, on the other hand, F$ contains a valid answer, the computer will simply proceed to line 140, which returns control to the main program section:

```
140 RETURN
```

For short, simple programs, you might be tempted to take a more direct approach to reading a yes-or-no answer:

```
100 INPUT "Y)ES OR N)O? "; A$
110 IF LEFT$(A$,1) <> "Y" THEN STOP
```

This sequence, which assumes that any response that does not begin with "Y" means *no*, is adequate when little is at risk. But in a long program—especially one that requires elaborate data input—it can be very annoying to terminate the performance accidentally by making a simple typing error when you meant to enter "Y" or "YES".

The "main program" section in Figure L.1, at lines 5 to 90, merely simulates the action of a program that depends on a yes-or-no response from the keyboard. Study the sample output, in Figure L.2, to see how the program reacts to a variety of answers.

LEN (string function)

The *function* LEN (the name stands for "length") requires a *string* argument; it returns an integer representing the length, in characters, of the string. For example, the expression:

```
LEN("HELLO")
```

would return the value 5, because HELLO contains 5 characters.

The argument of LEN may be expressed as a *literal* string value, as shown above, or as a string *variable* name:

```
LEN(S$)
```

LEN will also accept an argument that is a *concatenation* of two or more strings:

```
LEN(S$ + G$)
```

This expression will return the combined length of the two strings S$ and G$.

Sample Program

Figure L.3 shows a short program that uses LEN to center a string on the screen. The centering formula appears as part of a TAB function in line 70:

```
TAB((22 - LEN(S$))/2)
```

```
 10 PRINT CHR$(147)
 20 PRINT "ENTER ANY STRING:"
 30 PRINT: PRINT
 40 INPUT S$
 50 PRINT CHR$(147)
 60 GOSUB 150
 70 PRINT TAB((22-LEN(S$))/2);S$
 80 GOSUB 150
 90 GET X$: IF X$="" GOTO 90
100 IF X$<>"S" GOTO 10
110 END
150 FOR I = 1 TO 10: PRINT: NEXT I: RETURN
```

Figure L.3: LEN—Sample Program

This formula finds the difference between the length of the string, S$, and the width of the screen (i.e., 22 characters); this difference is divided by 2 to center the string. For the Commodore 64, which has a 40-character screen, you should change this expression to:

TAB((40 – LEN(S$))/2)

The program allows you to enter a string from the keyboard (line 40). It then displays this string in the center of the screen. So that you can experiment with other strings, the program loops back to the beginning after you press any key on the keyboard; to stop the program, press S (lines 90 and 100).

LET (command word)

The LET statement assigns a value to a *variable*. If the variable does not yet exist in the program, LET gives it an initial value. If the variable already exists, LET gives it a new value.

The LET statement takes the following form:

LET V = value

where V, on the left side of the equal sign, is any variable name (*string* or numeric). The value on the right side of the equal sign may be expressed as a *literal value*, a variable, or an expression composed of literals and/or variables. The LET statement instructs the computer to evaluate the expression to the right of the equal sign and to store the resulting value in the memory location represented by the variable name to the left of the equal sign.

Here are three examples, paraphrased:

```
LET AGE = 18
```

"Store the value 18 in the variable AGE."

```
LET I = J
```

"Store the value of the variable J in the variable I." (The variable J should be assigned a value in advance of this statement. The value of J does *not* change as a result of this statement.)

```
LET N = 5 * M + P/2
```

"Evaluate the expression on the right side of the equal sign, and store the resulting value in the variable N." (The variables M and P are assumed to contain values at the time the LET statement is performed. The values of M and P do *not* change as a result of the statement.)

Notice that while there is no practical limit to the complexity of the expression on the right side of the equal sign, there is never more than a single variable name on the left side of the equal sign.

If you refer to a numeric variable that has not yet explicitly been assigned a value, the variable automatically receives the value zero. A string variable that has not yet been assigned a value is empty.

The LET statement may be performed either as an *immediate command* or as a program instruction.

Sample Program

The program shown in Figure L.4 is an exercise that demonstrates several different uses of the LET statement. The statements in lines 20 to 60 assign string values to the five elements of the string array L$. Notice that in each of these LET statements the variable name on the left side of the equal sign is actually the name of an array element.

Lines 70, 130, and 140 work together to simulate the action of a FOR loop in this program. Line 70 initializes the variable I to the value 1. This variable will be used as a counter in the loop. The LET statement in line 130 increments the value of I by 1 for each repetition of the loop:

```
130 LET I = I + 1
```

This kind of statement often proves mysterious to the beginning programmer; it can be paraphrased as follows: "Add 1 to the current value of I; then store the new, incremented, value back again in I." The old value of I is lost.

The LET statement in line 90 determines the horizontal tab location of each message that will be displayed on the screen:

```
 90 LET H = (I - 1) * 3
100 PRINT : PRINT
110 PRINT TAB(H);
```

Remember that this LET statement does not change the value of the variable I. Only the variable H receives a new value.

Study the output from this program (Figure L.5) carefully. Make sure you understand how the LET statement in line 130 controls the action of the loop that creates the screen display.

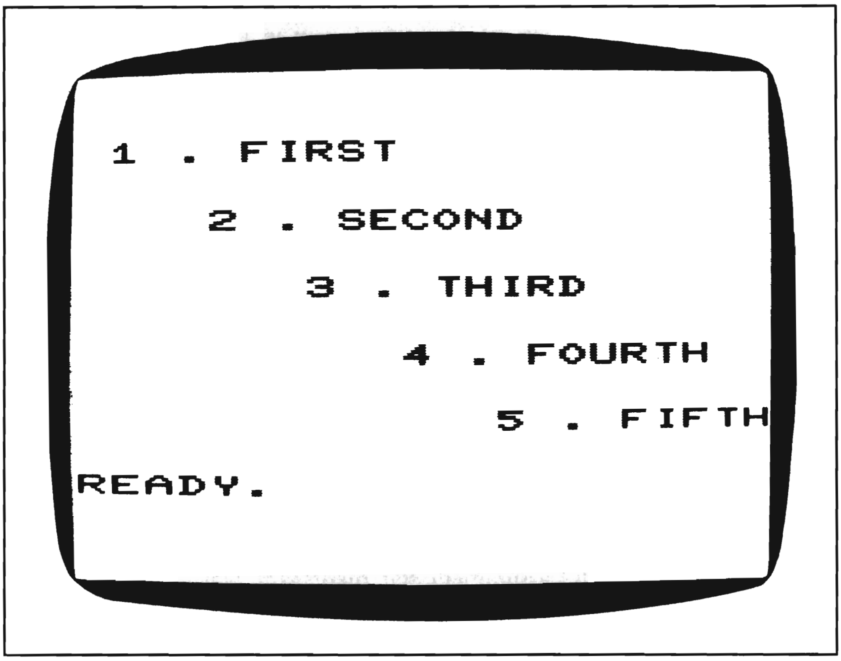
Notes and Comments

— In BASIC assignment statements, the word LET is actually optional. Thus, you may see statements such as:

```
130 I = I + 1
```

```
  5 PRINT CHR$(147)
 10 DIM L$(5)
 20 LET L$(1)="FIRST"
 30 LET L$(2)="SECOND"
 40 LET L$(3)="THIRD"
 50 LET L$(4)="FOURTH"
 60 LET L$(5)="FIFTH"
 70 LET I = 1
 80 REM ** START LOOP
 90 LET H = (I-1)*3
100 PRINT : PRINT
110 PRINT TAB(H);
120 PRINT I;". ";L$(I)
130 LET I = I + 1
140 IF I<=5 GOTO 80
150 END
```

Figure L.4: LET—Sample Program



```
1 . FIRST
    2 . SECOND
        3 . THIRD
            4 . FOURTH
                5 . FIFTH
READY.
```

Figure L.5: LET—Sample Output

in some BASIC program listings. The advantage of using the word LET is simply that it enhances clarity; its use is a matter of personal preference. In this book, all assignment statements begin with LET.

LIST (command word)

The LIST command instructs the computer to display on the screen the lines of the program currently stored in its memory. LIST is almost always performed as an *immediate command.* The command may take one of several forms. The first is simply:

```
LIST
```

This command results in a listing display starting from the first line of the program and continuing to the end of the program. The second form of the LIST command is:

```
LIST L
```

where L is a value that represents a line number in the program. In this case the computer displays only line L on the screen.

Finally, BASIC also allows the form:

```
LIST L1-L2
```

where L1 and L2 are both literal numeric values representing line numbers in the program. The result of this command is to display the portion of the program from line L1 to line L2.

Two variations on this final form are available. To list the program from the beginning up to line L2, you can give the command:

```
LIST -L2
```

and to list the program from line L1 to the end, you can type:

```
LIST L1-
```

Notes and Comments

— If a program listing takes up more than one screen, the computer simply scrolls the screen display down to the end of the program. (This means that once the screen is full, the line at the top will disappear, and each subsequent line will move up by one row. The next line of the program will appear at the bottom of the screen.) To slow down this scrolling action, you can hold down on the CTRL-key while the program is listing.

Literal Value (computer vocabulary)

A literal value is an actual numeric or *string* value, entered as a constant in a program statement; as opposed to a *variable* name, which simply represents the numeric or string value that is currently stored in the computer's active memory under that name. A literal string value must appear within quotation marks in a program instruction; for example, the following statement assigns a literal string value to the variable S$:

```
LET S$ = "COMPUTER"
```

A real number may appear in either decimal form or *scientific notation,* as in the following examples:

```
LET N1 = 123.456
LET N2 = 3.1 E+8
```

LOAD (cassette or disk command)

The LOAD command retrieves a BASIC program from a disk or cassette file and places the program in the computer's active memory. After LOADing, the program is ready to run.

The LOAD command takes a variety of formats; probably the most common and general one is:

```
LOAD "FILE NAME", D
```

where the file name, entered between quotes (or as a string variable value), may be from 1 to 16 characters long; and D represents the device number. For a cassette tape recorder, D is 1; for a disk drive, the device number is 8.

For a cassette file, the device number is optional in the LOAD command:

```
LOAD "FILE NAME"
```

When you enter this command, if you have not yet turned on the cassette recorder, the computer will display the message:

```
PRESS PLAY ON TAPE
```

Once you have followed this direction, the computer will play the recorder until it finds the program named in the LOAD command. When the desired program has been loaded, the computer automatically stops the operation of the cassette recorder. You can also use the command:

```
LOAD
```

for loading a program from a cassette, in which case the first complete program that the computer encounters will be loaded into memory.

For a disk drive, you must use the complete form of the LOAD command:

```
LOAD "FILE NAME", 8
```

If the program you have named is not on the disk, or if the name refers to a file that is not a program file, the computer will display the following *error message:*

```
? FILE NOT FOUND
  ERROR
```

Remember that LOAD is just for loading program files, not data files. The one exception is the disk directory, a list of all the files (programs or otherwise) stored on the disk. You can use the LOAD command to load the directory into the computer's memory, as follows:

```
LOAD"$",8
```

Once the directory is present, you can type LIST to display it on the screen. A word of caution: whenever you load a program—or the disk directory—into the computer's memory, you lose any program that was there before. You should make sure you save the current program before you use the LOAD command.

Notes and Comments

— The LOAD command has one further optional parameter that you may not use very often, but should be aware of all the same; it is the memory address parameter, M:

```
LOAD "FILE NAME", D, M
```

If M is zero, or if M is not part of the command, the program will be loaded into memory starting at the beginning address of the area reserved for BASIC programs. If M is 1, the program will be loaded into the same memory location that it was originally SAVEd from.

LOG (function)

The LOG *function* supplies the natural logarithm (base *e*) of a number. The *argument* of LOG must be greater than 0.

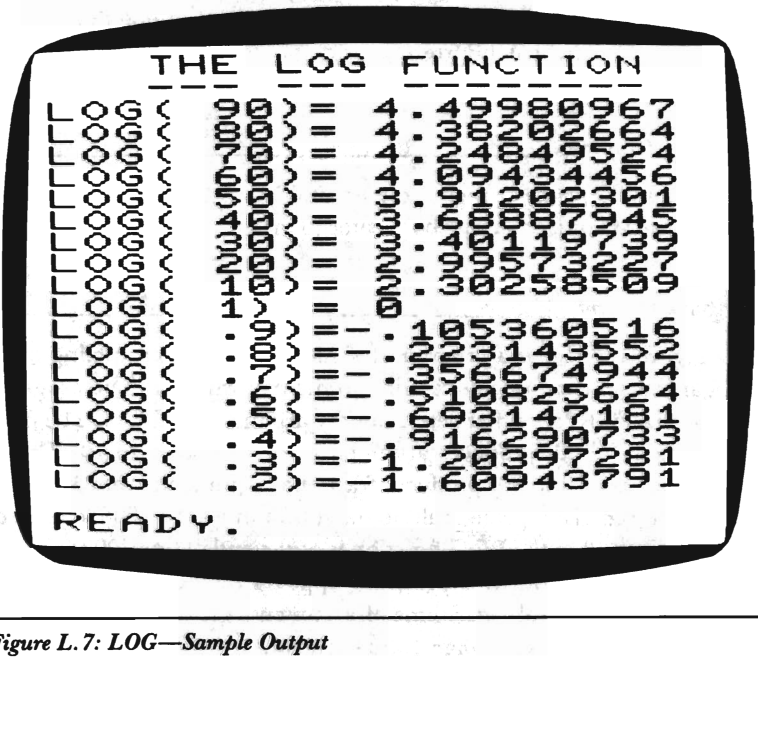
Sample Program

Figure L.6 shows a program designed to display a sampling of natural logarithms for arguments ranging from 90 down to .1. The program contains two FOR loops that determine the arguments of LOG. The first loop, at lines 40 to 60, produces arguments from 90 down to 10. The second loop, at lines 70 to 90, produces fractional arguments from 1 down to .1. Both loops make repeated calls to the subroutine at line 200 to print output lines. The call to the LOG function is at the end of line 200.

The output from this program appears in Figure L.7. Notice that whereas the natural logarithms of arguments greater than 1 are positive, those of arguments less than 1 are negative.

```
 10 PRINT CHR$(147)
 20 PRINT "   THE LOG FUNCTION"
 30 PRINT "   --- --- --------"
 40 FOR I = 90 TO 10 STEP -10
 50   GOSUB 200
 60 NEXT I
 70 FOR I = 1 TO .1 STEP -.1
 80   GOSUB 200
 90 NEXT I
100 END
200 PRINT "LOG(";STR$(I);")";TAB(8);"=";LOG(I)
210 RETURN
```

Figure L.6: LOG—Sample Program



```
    THE LOG FUNCTION
    --- --- --------
LOG( 90)= 4.49980967
LOG( 80)= 4.38202664
LOG( 70)= 4.24849524
LOG( 60)= 4.09434456
LOG( 50)= 3.91202301
LOG( 40)= 3.68887945
LOG( 30)= 3.40119739
LOG( 20)= 2.99573227
LOG( 10)= 2.30258509
LOG( 1) = 0
LOG( .9)=-.105360516
LOG( .8)=-.223143552
LOG( .7)=-.356674944
LOG( .6)=-.510825624
LOG( .5)=-.693147181
LOG( .4)=-.916290733
LOG( .3)=-1.20397281
LOG( .2)=-1.60943791

READY.
```

Figure L.7: LOG—Sample Output

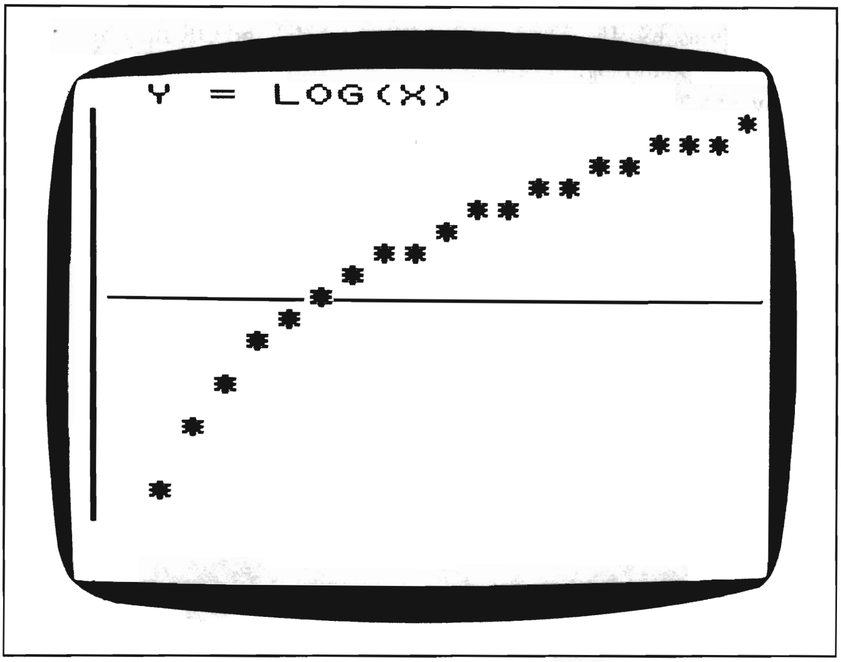


Figure L.8: LOG—Plotted Graph

Notes and Comments

— Figure L.8 shows a crude plotted graph of the LOG function, produced on the VIC-20. The curve represents the equation:

$y = \log_e x$

The curve crosses the x-axis at (1,0).

— Arguments of zero or values less than zero are illegal for LOG, and result in the following error message:

```
? ILLEGAL QUANTITY
  ERROR
```

— *See also* the entry under EXP.

Logical Expression (computer vocabulary)

A logical expression is one that the computer evaluates as either true or false. Logical expressions typically take the form of equalities or inequalities. (*See* IF.) The logical operators AND and OR may be used to build compound logical expressions. The logical *function* NOT negates the value of a logical expression.

A simple numeric value may take the place of a logical expression in an IF statement. If the *variable* contains the value zero, it will be evaluated as false; if it contains any other value, it will be evaluated as true.

Machine Code (computer vocabulary)

Machine code consists of instructions that a specific microprocessor can perform directly, rather than those in a language that must be interpreted, such as BASIC. Writing programs in machine code for the Commodore computers requires an understanding of the instruction sets and architectures of the appropriate microprocessors—the 6502 for the VIC-20, and the 6510 for the Commodore 64. BASIC has commands that allow you to examine or change the contents of a memory location (PEEK and POKE); and to "call" a machine code subroutine during the performance of a BASIC program (SYS and USR). Machine code is also known as "machine language," or "native code."

Menu (computer vocabulary)

A menu is a display of the options available to the user at a given point in a program performance. The menu must also indicate an unambiguous method of choosing one of the options. (*See the entry under* GOSUB.)

MID$ (string function)

The MID$ *function* accesses a specified sequential portion of a *string*. The function takes the form:

MID$(S$,P,N)

where S$ is a string and P and N are integers. (S$ may be expressed as a *literal* string, a string *variable*, or a string expression.) MID$ returns N characters of S$, starting from the Pth character in the string. For example, in the following statement:

```
PRINT MID$("COMPUTER",4,3)
```

MID$ returns 3 characters of the string COMPUTER, starting from the 4th character. The statement will thus display the word PUT on the screen.

Notes and Comments

— The program listed and described under the heading STR$ shows the MID$ function in action. That program simulates the PRINT USING instruction, which is missing in the Commodore versions of BASIC.

— *See also the entries under* LEFT$ and RIGHT$.

NEW (command word)

The NEW command effectively erases the current program from the computer's memory. After you have entered NEW, you cannot retrieve the program unless you have first stored it on disk or cassette tape.

NEXT (command word)

The NEXT statement marks the end of a sequence of program lines that make up a FOR loop. The most straightforward form of the NEXT statement is:

```
NEXT V
```

where V is the control variable established in the FOR statement that introduces the loop. (*See the entry under* FOR.) For the sake of programming clarity, this is probably the best form of the NEXT statement to use, although other forms are available.

The variable name may be omitted from the NEXT statement, as in the following example:

```
10 FOR I = 1 TO 10
20    PRINT I
30 NEXT
```

Furthermore, BASIC allows a single NEXT statement to mark the end of a series of nested loops; for example:

```
 10 FOR I = 1 TO 10
 20 FOR J = 1 TO 5
 30 FOR K = 1 TO 20
    . . .
100 NEXT K, J, I
```

Notice the order of the control variables in this NEXT statement: from the innermost to the outermost loop. If this order is written incorrectly—or if any NEXT statement references an incorrect control variable—the program will terminate with an error message:

```
? NEXT WITHOUT FOR
  ERROR IN 100
```

NOT (logical operator)

The logical operator NOT modifies a *logical expression* in an IF statement; it reverses the value of the expression it modifies:

— If a logical expression is true, NOT makes it false.
— If a logical expression is false, NOT makes it true.

NOT must always appear immediately before the expression it modifies:

```
IF NOT (logical expression) THEN (command)
```

Sample Program

The program under the heading LEFT$ (Figure L.1) contains an interesting example of NOT. It is used in a passage that validates what should be a yes-or-no input response:

```
110 INPUT "Y)ES OR N)O";A$
120 LET F$ = LEFT$(A$,1)
130 IF NOT (F$ = "Y" OR F$ = "N") GOTO 150
```

Line 130, paraphrased, says: Send control of the program to line 150 if neither of the following two statements is true:

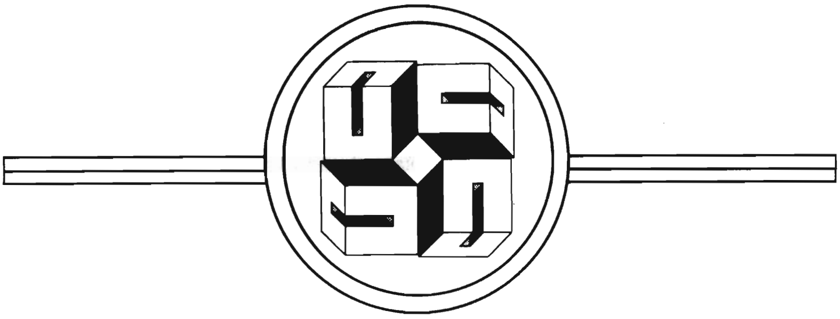
```
F$ = "Y"
F$ = "N"
```

Thus, if the first character of the input string F$ is something other than "Y" or "N", the contingency statements starting at line 150 are performed.

IF statements that contain NOT can always be rewritten to eliminate NOT. For example, line 130 of the sample program could have appeared as:

```
130 IF F$<>"Y" AND F$<>"N" THEN GOTO 150
```

The computer's action would be the same for both versions of the line. However, from the point of view of a person who is reading the program listing for the first time, the version with NOT is probably easier to understand. The use of NOT, then, is largely a matter of programming clarity and style.



ON (command word)

ON provides a means of choosing among a list of program lines in GOSUB and GOTO statements. The statement can take the forms:

ON N GOSUB (list of subroutine starting lines)

or:

ON N GOTO (list of program lines)

where N is any numeric variable or arithmetic expression that evaluates to a positive number. The computer uses the integral value of this number to choose one of the program lines in the list. The best way to see how ON works is to look at an example:

```
10 ON I GOTO 150, 200, 250, 300, 350
```

This ON . . . GOTO statement contains a list of five program line numbers—150, 200, 250, 300, and 350. Depending on the value of I, control of the program can be sent to any one of these lines. If I equals 1, control will be sent to the first line in the list, line 100; if I equals 2, control will be sent to the second line, 200; and so on. If I equals any value from 1 to 5, one of the five line numbers will be chosen for transfer of control.

You can see how economical the ON statement is; the example above takes the place of five IF statements:

```
11 IF I = 1 GOTO 150
12 IF I = 2 GOTO 200
13 IF I = 3 GOTO 250
14 IF I = 4 GOTO 300
15 IF I = 5 GOTO 350
```

Since I is not necessarily an integer, an additional statement is also implied by the ON statement. The computer finds the integral value of I before choosing one of the line numbers in the list:

```
10 LET I = INT(I)
```

If the integral value of the variable named after ON is greater than the number of lines in the list, the ON statement results in no action. (For example, if I equals 6 in the ON statement above, no transfer of control will occur.) Likewise, if the integral value of the variable is zero, no action will result. For this reason, you will usually want to test the value of the variable before the performance of the ON statement, as in the following sequence:

```
10 INPUT I
20 IF I<1 OR I>5 GOTO 10
30 ON I GOSUB 100,200,300,400,500
```

Line 10 reads an input value from the keyboard. If the value of I is outside the range that will produce action in the ON . . . GOSUB statement, line 20 loops back for another input value. Only when an appropriate value is input for I will the ON . . . GOSUB statement be performed.

If the value of the variable named after ON is less than zero, the program will terminate with an error message similar to the following:

```
? ILLEGAL QUANTITY
  ERROR IN 30
```

In this instance, 30 is the line number of the offending ON statement.

OPEN (input/output command word)

The OPEN command opens an external file for reading or writing. An OPEN command precedes each INPUT#, PRINT#, GET#, and CMD command. OPEN specifies many things about the file and the way the computer will handle the file: the file number, F, by which the computer will identify the file's "buffer"; the device number, D, indicating the device location of the file; the secondary address, S, which may indicate how the file will be accessed; the name of the file; and the file type. Because OPEN has so many possible parameters, which vary in so many ways, it is one of the Commodore computers' most confusing commands. However, you will probably use OPEN most often for a few common operations involving data storage on cassette or disk; the formats of OPEN for these few operations are not hard to master.

The general format of the OPEN command for data file storage is:

```
OPEN F, D, S, "FILE NAME"
```

You can choose any number from 1 to 127 to act as the file number, F, for each file you open. The device number, D, is 1 for the cassette, or 8 for the disk. The secondary address, S, varies according to the device and the situation.

Let's look at some examples. Say you want to create a data file named EMPLOYEES on cassette; in other words, you are going to open a file and write some data to it. You have no other files open at the same time, so you will simply give the file the number 1. You can use the following OPEN command:

```
OPEN 1,1,1, "EMPLOYEES"
```

In this case, the first 1 is the file number, which you have chosen yourself. The second 1 is the device number, specifying the cassette. The third 1, the secondary address, here indicates that you are opening the file for writing. After you have performed this command, you can send data to the file using the PRINT# command; for example:

```
PRINT#1, "SMITH", 1800
```

The number 1 in the PRINT# command refers to the file number, F, the first parameter of the OPEN command.

Let's say you have written some data to this file, and subsequently closed the file, with the command:

```
CLOSE 1
```

Later, you want to reopen the file and read the data that is stored in it. You can use almost the same OPEN command that you gave to open the file for writing; the only change will be in the secondary address. To open a cassette file for reading, you must use a secondary address of 0:

```
OPEN 1,1,0, "EMPLOYEES"
```

After this command you may use INPUT# commands to read the data stored in the file; for example:

```
INPUT#1, N$, S
```

Opening data files on disk is only slightly more complicated. Recall that the device number for the disk is 8. The secondary address may be any number from 2 to 14. Whenever you have more than one file open at once, each file should have its own secondary address; beyond that, the secondary address has no particular meaning to you.

Along with the name of the file, inside quotation marks, you must indicate some further information about the file. If you have two disk drives you must indicate which drive should be activated for finding the file; the drives are numbered 0 and 1. You must also indicate the type of data file: S for sequential file; R for relative file. Finally, for sequential files, you must indicate whether the file is being opened for writing (W) or reading (R). The following command opens a sequential file named EMPLOYEES, located on drive 0, for writing:

```
OPEN 1,8,2, "0:EMPLOYEES,S,W"
```

In this case, 1 has been assigned as the file number, and 2 as the secondary address. If you have only one drive, you can use the simpler form:

```
OPEN 1,8,2, "EMPLOYEES,S,W"
```

To open the same file for reading, the command would be:

```
OPEN 1,8,2, "EMPLOYEES,S,R"
```

The sample programs under the entries GET#, INPUT#, and PRINT# show examples of the OPEN command.

Notes and Comments

— The OPEN command can also be used to perform a number of very useful disk commands. For example, you can copy a file, rename a file, remove a file, or format a new diskette. The secondary address 15 must be used for all these commands; we might consider the following a general format for performing these commands:

```
OPEN 1,8,15, "COMMAND"
```

Here, 1 is specified as the file number; you can of course use any file number you want. Once a file number is opened with the secondary address 15, you can also use the PRINT# statement to perform disk commands:

```
PRINT #1, "COMMAND"
```

What follows is a summary of some of the most useful commands available:

The COPY command. This command makes a copy of a file; the copy will be made on the same drive as the original.

For example, the following command makes a copy of the file named OLD; the copy will be named NEW:

```
OPEN 1,8,15, "C:NEW=OLD"
```

The RENAME command. This command gives a new name to any disk file:

```
OPEN 1,8,15,"R:NEW NAME=OLD NAME"
```

The SCRATCH command. This command removes a file from the disk. After you have used this command, there is no way to recover the file:

```
OPEN 1,8,15,"S:FILE NAME"
```

The NEW command. This command initializes a new diskette. You must use this command before the computer will store any information on a diskette.

```
OPEN 1,8,15,"N:DISK NAME,ID"
```

ID can be any two characters, which will serve to identify the diskette. The DISK NAME that you decide on will appear in the disk directory.

In all these examples, an abbreviated form of each command has been used. Note that you may also use the full name of each command; for example:

```
OPEN 1,8,15,"SCRATCH:FILE NAME"
```

OR (logical operator)

The logical operator OR can be used to create a compound *logical expression* for an IF decision. The value of such a compound expression depends on the values of the elements that are combined by OR. A compound expression in the following form:

```
statement-1 OR statement-2
```

is true if either statement-1 or statement-2 is true, or if both statements are true. If both statements are false, the compound expression is also false.

Sample Program

The sample program listed and described under the AND entry (Figure A.4) also contains an example of the use of OR. Line 190 of the program tests the values of the two *variables* AVE (for "average quiz score") and F (for "final exam score"):

```
190 IF AVE < 75 OR F < 70 THEN PRINT "FAILED"
```

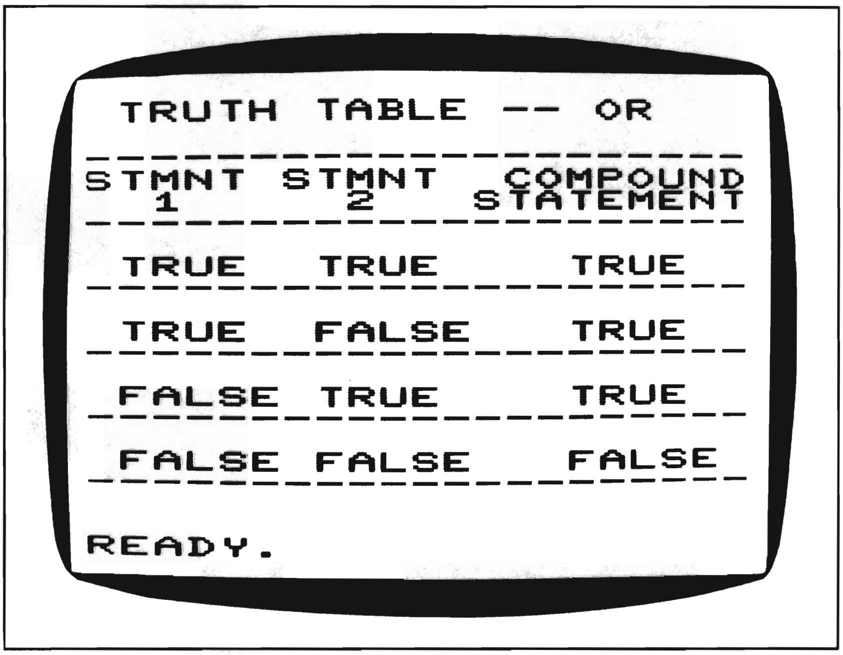
If either of the scores falls below the cut-off point (75 for AVE; 70 for F), the compound expression:

```
AVE < 75 OR F < 70
```

will be evaluated as true, resulting in the message "FAILED" appearing on the screen. The compound expression will be evaluated as false only if both of its elements are false; that is, if both scores are at the passing point or better.

Notes and Comments

— Figure O.1 is a "truth table" for OR conditions. It shows the resulting value of a compound expression, given different combinations of values for statement-1 and statement-2. Notice



```
  TRUTH  TABLE  --  OR
-----------------------
STMNT  STMNT   COMPOUND
  1      2    STATEMENT
-----------------------
 TRUE   TRUE      TRUE
-----------------------
 TRUE   FALSE     TRUE
-----------------------
 FALSE  TRUE      TRUE
-----------------------
 FALSE  FALSE     FALSE
-----------------------

READY.
```

Figure O.1: OR—Truth Table

that the compound expression is true in three cases—when either one of the inner statements is true, or when both are true.

— *See the entries under* AND, IF, and NOT for more information.

PEEK (function)

The PEEK *function* supplies the contents of a specified memory location, in decimal form. The PEEK statement takes the following form:

```
PEEK(M)
```

where M is a *literal* numeric *value*, *variable*, or *arithmetic expression* that represents a memory location numbered 0 to 65535. PEEK returns a decimal number from 0 to 255, the contents of one *byte* of memory. Since PEEK is a function, it cannot, of course, stand alone in a program, but must be part of a statement that makes use of the value it returns—for example, PRINT or LET.

The PEEK and POKE statements, which together supply a much more intimate access to the computer's inner organization than do other BASIC commands, are useful for programmers who wish to write *machine-code* routines. Such routines require a knowledge of the machine-code instruction set of the appropriate microprocessor—6502 for the VIC-20, or 6510 for the Commodore 64—that is the central processing unit of the computer. (*See the entries under* POKE, USR, and SYS for more information.)

Sample Program

Figure P.1 shows a program illustrating the use of PEEK. The program is an exercise designed to look into the memory addresses where the program itself is stored and display the first fifty or so bytes of the program.

The first line of the program is a REM line; it is this line that we will locate in the computer's memory:

```
10 REM ** LOCATE THIS LINE IN THE COMPUTER'S
          MEMORY
```

The VIC-20 computer stores BASIC programs beginning at memory location 4096; this program contains two FOR loops (at lines 30 and 70) that PEEK through addresses 4096 to 4150; for example:

```
30 FOR I = 4096 TO 4150
```

The Commodore 64 computer stores BASIC programs beginning at address 2048, so if you want to run this program on that computer, you'll have to revise lines 30 and 70 accordingly.

The first FOR loop simply PEEKs into each location and prints the decimal values stored in those locations:

```
40 PRINT PEEK(I); " ";
```

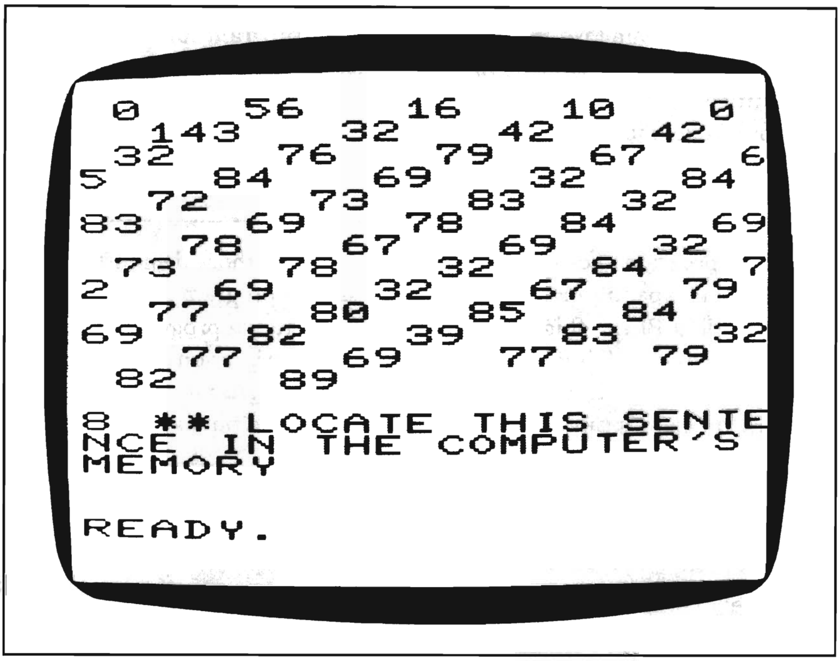
This loop will thus display a series of decimal numbers between the values 0 and 255. The second FOR loop uses the CHR$ function to translate these values into their ASCII code equivalents, and displays those equivalents on the screen:

```
80 PRINT CHR$(PEEK(I)); " ";
```

The result of this loop, then, will be something that you can actually recognize. Figure P.2 shows the output from this program; you can see the results of the two FOR loops. Look carefully at the string of numbers produced by the first loop. The ASCII code for the character "L" is 76. If you count forward from the first value, you will see that the word LOCATE is stored

```
 10 REM ** LOCATE THIS SENTENCE IN THE COMPUTER'S MEMORY
 20 PRINT CHR$(147)
 30 FOR I = 4096 TO 4150
 40   PRINT PEEK(I); " ";
 50 NEXT I
 60 PRINT : PRINT
 70 FOR I = 4096 TO 4150
 80   PRINT CHR$(PEEK(I));
 90 NEXT I
100 PRINT : PRINT
```

Figure P.1: PEEK—Sample Program



```
 0   56   16   10   0
  143   32   42   42
 32   76   79   67   6
5   84   69   32   84
  72   73   83   32
83   69   78   84   69
   78   67   69   32
 73   78   32   84   7
2   69   32   67   79
  77   80   85   84
69   82   39   83   32
   77   69   77   79
 82   89

8 ** LOCATE THIS SENTE
NCE IN THE COMPUTER'S
MEMORY

READY.
```

Figure P.2: PEEK—Sample Output

beginning at memory location 4106. We will make use of this piece of information in a similar program exercise under the heading POKE.

POKE (command word)

The POKE command writes a value into a specified memory location. The POKE statement is written as follows:

POKE M, V

where M is a memory location from 0 to 65535, and V is the value to be written into the memory location. V must be in the range:

0 < = V < = +255

Both M and V may be expressed as *literal* numeric *values*, *variables*, or *arithmetic expressions*.

POKE can be used to store *machine-code* instructions in the computer's memory. The instructions of the microprocessor—6502 for the VIC-20; 6510 for the Commodore 64—are all coded in numbers from 0 to 255. It is sometimes useful to write a sequence of machine-code instructions that will perform a certain task at some point during a program run. Such instructions can be POKEd into memory and then performed via the USR or SYS commands. Accomplishing this, however, requires a knowledge of the appropriate microprocessor's instruction set.

Sample Program

The program shown in Figure P.3 provides a short demonstration of POKE; it expands upon elements of the sample program described under the heading PEEK (Figure P.1). This demonstration program is carefully designed to POKE information only into memory locations where no harm can be done. POKE is not a command to be used gratuitously. Before you revise memory locations for whatever reasons, find out exactly what parts

```
 10 REM ** LOCATE THIS SENTENCE IN THE COMPUTER'S MEMORY
 20 PRINT CHR$(147)
 25 PRINT "==> PEEK AT MEMORY.": PRINT
 30 GOSUB 200
 35 PRINT : PRINT "==> POKE 'REVISE'."
 40 LET S$="REVISE"
 50 LET M = 4105
 60 FOR I = 1 TO 6
 70   POKE M+I, ASC(MID$(S$,I,1))
 80 NEXT I
 90 PRINT
 95 PRINT "==> TAKE ANOTHER PEEK."
100 GOSUB 200
110 PRINT
115 PRINT "==> LIST LINE 10."
120 LIST 10
130 PRINT
140 END
200 FOR I = 4096 TO 4150
210   PRINT CHR$(PEEK(I));
220 NEXT I
230 PRINT
240 RETURN
```

Figure P.3: POKE—Sample Program

of memory are currently available. Indiscriminate POKEs into memory can destroy the program you are currently working on, or make it necessary to reboot the system. (You can't, of course, do any *permanent* damage to your computer with the POKE command, but you can cause temporary problems.) Also note that this program, in its current version, is designed for the VIC-20 computer; if you want to try it on the Commodore 64 computer, you'll have to revise the memory addresses. (*See* PEEK.)

Like the sample program under the heading PEEK, this program begins with a REM line; it is this REM line that will be changed by the program's POKE commands:

```
10 REM ** LOCATE THIS SENTENCE IN THE COMPUTER'S
          MEMORY
```

When we ran the PEEK program, we discovered that this sentence was stored in memory locations beginning at address 4106. Now we will see how POKE is used to change the value stored in a memory location. If you look at the output from the POKE program (Figure P.4), you will see the four activities the program performs:

1. It PEEKs at the memory locations that store the program's REM line, and it displays the character equivalents of these memory values on the screen.
2. It then POKEs six new characters—the letters of the word REVISE—into memory locations 4106 to 4111, the addresses that formerly held the word LOCATE.
3. It once again PEEKs at the memory locations of the beginning of the program, and displays the new version of the REM line on the screen. You can see in Figure P.4 that the line has indeed been revised.
4. It LISTs line 10 of the program on the screen to demonstrate that the program itself has been revised by the POKE commands.

The subroutine at line 200 performs the PEEK operation; this subroutine is called twice from the "main program" section. The revision of the REM line is accomplished by lines 40 to 80. The string variable S$ contains the six characters to be POKEd into memory, and the variable M contains the address of the memory location just before the ones that will be revised:

```
40 LET S$ = "REVISE"
50 LET M = 4105
```

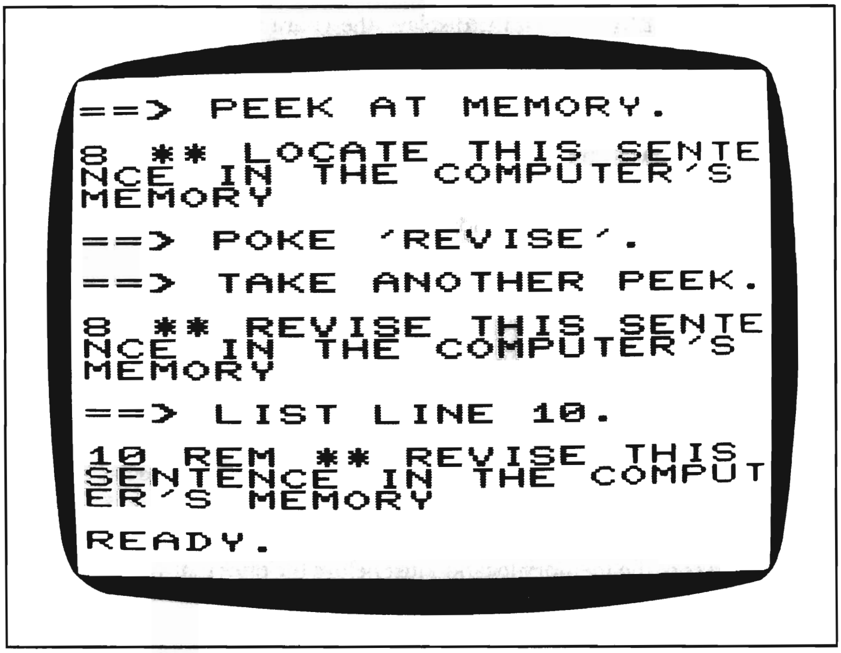
Inside the FOR loop at line 60 is a single instruction, the POKE statement:

```
60 FOR I = 1 TO 6
70    POKE M + I, ASC(MID$(S$,I,1))
80 NEXT I
```

Starting at memory location 4106 (i.e., M + 1), each character of S$ is accessed one at a time (via the MID$ function), converted to its ASCII decimal equivalent (via the ASC function), and stored in the computer's memory (via the POKE statement). That is all this FOR loop does. Notice that the control variable of the FOR loop is used twice in the POKE statement: first to determine the correct memory location for each POKE, then to specify the position of the current character that is to be accessed from the string S$.

Notes and Comments

— The sample program described under the heading GET shows a useful example of the POKE statement. Both the VIC-20 and



```
==> PEEK AT MEMORY.
8 ** LOCATE THIS SENTE
NCE IN THE COMPUTER'S
MEMORY
==> POKE 'REVISE'.
==> TAKE ANOTHER PEEK.
8 ** REVISE THIS SENTE
NCE IN THE COMPUTER'S
MEMORY
==> LIST LINE 10.
10 REM ** REVISE THIS
SENTENCE IN THE COMPUT
ER'S MEMORY
READY.
```

Figure P.4: POKE—Sample Output

the Commodore 64 computers have memory locations that control the colors of the background and the frame on the display screen. You can change these colors by POKEing appropriate values into their memory locations. On the VIC-20, the single memory location at address 36879 controls the colors of both the background and the frame. You can use the following POKE statement to change the colors:

```
POKE 36879, 16 * B + F - 9
```

where the variable B and F contain integer values from 1 to 8, representing the desired colors of the background and the frame, respectively. On the Commodore 64 computer, memory location 53280 controls the frame color, and location 53281 controls the background color. The following POKE commands, then, will change the colors:

```
POKE 53280, F - 1
POKE 53281, B - 1
```

— *See also the entries under* PEEK, USR, and SYS.

POS (function)

The POS *function* returns an integer representing the current horizontal position of the *cursor*. The format of POS is:

```
POS(N)
```

The *argument*, N, may be any valid numeric value; it has no significance in the function.

PRINT (command word)

The PRINT command displays information on the video screen or other output device. A single PRINT statement may contain many elements to be displayed, including: *literal values* (both numbers and *strings*); the values of numeric or string *variables*; the results of numeric, string or *logical expressions*; and graphics characters and control characters. In addition, the PRINT statement provides techniques for arranging tabular data on the screen.

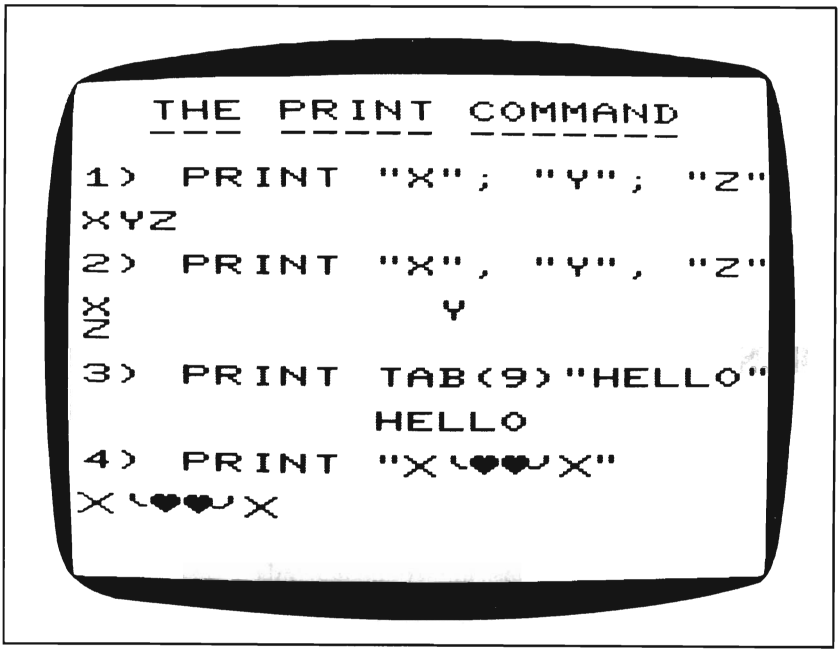
Figures P.5 and P.6 show a series of eight examples of the PRINT command and the screen display output lines resulting from these commands. These examples illustrate the range of techniques available with PRINT.

The following notes discuss each example, from 1 to 8. If you wish to try these examples on your computer, all of them may be entered as immediate commands. Notice that some of the examples contain more than one command. BASIC uses the colon character (:) to separate multiple commands on a single line.

1. Use of the semicolon. When a semicolon separates two string elements of a PRINT command, the elements will be displayed on the screen side-by-side with no space separating them, as shown in this example. Furthermore, if a PRINT statement ends in a semicolon, any subsequent PRINT statement will begin its display where the previous display left off. For example, consider the following lines:

```
 5 LET J$ = "JACK"
10 PRINT "HELLO ";
15 PRINT J$
```

The semicolon at the end of line 10 prevents the computer from moving to a new line for a subsequent PRINT command.



```
    THE PRINT COMMAND
    --- ----- -------
1)  PRINT  "X";  "Y";  "Z"
XYZ
2)  PRINT  "X",  "Y",  "Z"
X              Y
Z
3)  PRINT  TAB(9)"HELLO"
           HELLO
4)  PRINT  "X╰♥♥╯X"
X╰♥♥╯X
```

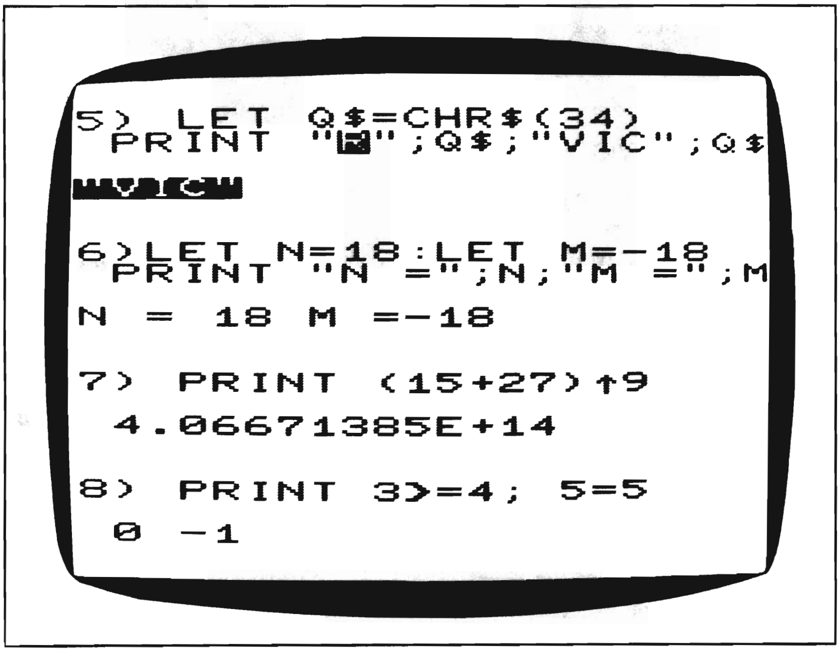
Figure P.5: PRINT—Examples

Thus, the result of this sequence will be:

HELLO JACK

2. ***Use of the comma.*** A comma separating two elements of a PRINT statement will cause a tab forward to a pre-set tab stop on the current display line or to the beginning of the next display line, depending on the current position of the cursor. The Commodore 64 has a 40-column screen, with tab stops at columns 11, 21, and 31. The VIC-20, with its 22-column screen, has one tab stop at column 11; in the example you can see that the first comma places the Y at column 11 and the second comma places the Z at the beginning of the next display line.

3. ***Use of the TAB function.*** The argument of TAB indicates the column number where the next display element will begin in the current display line. TAB(9) in this example means that the "H" of HELLO will appear in column 9, with the rest of the characters following. *(See the entry under* TAB.)



```
5)  LET  Q$=CHR$(34)
 PRINT  "■";Q$;"VIC";Q$
"VIC"

6)LET N=18:LET M=-18
 PRINT  "N =";N;"M =";M
N =  18  M =-18

7)  PRINT  (15+27)↑9
 4.0667138E+14

8)  PRINT  3>=4;  5=5
 0  -1
```

Figure P.6: PRINT—Examples (Continued)

4. Printing graphics characters. All of the letter keys on the computer's keyboard (plus several of the symbol keys) display pairs of graphics characters. These graphics characters can be entered directly into a PRINT statement from the keyboard. Each pair of characters appears side-by-side on the front of a key. To enter the character on the right, press the SHIFT key and the letter key at the same time; to enter the character on the left, press the "COMMODORE" key (the key located at the lower-left corner of the keyboard) and the letter key together. The graphics characters shown in the example are all SHIFT characters, entered from the letter keys V, J, S, S, K, V, respectively.

5. Printing the quotation-mark character. The only way to PRINT a quotation-mark is via a reference to its ASCII character-code number. Since the quote character is 34 in ASCII, the following statement assigns this character to the variable Q$:

```
LET Q$ = CHR$(34)
```

Then Q$ can be used in the following manner to display quotation marks on the screen:

```
PRINT Q$; "VIC"; Q$
```

This statement results in the following output:

```
"VIC"
```

Printing control characters. As illustrated in the fifth PRINT example (top of Figure P.6), control characters may be activated from within PRINT commands. The control characters include keys that change the text color; keys that control cursor movement; and, in this case, the reverse-video on and off keys. To activate any of these control functions from within a PRINT statement, you can type the appropriate keys inside quotation marks. On the screen, in a PRINT statement itself, the control character will appear as an arbitrary, and usually meaningless, graphics character; but when the PRINT statement is performed, the computer will read this character as a command to perform the control function it represents. In the example, the reverse-video R character that appears within quotation marks at the beginning of the PRINT statement is a result of pressing the CTRL key and the 9 key at the same time. When the PRINT statement is performed, this character activates the reverse-video display function; as you can see, the resulting output line is printed in reverse-video characters.

An alternative technique for activating control functions is to use the CHR$ function to print the ASCII code equivalent of the control character. (This technique is described in detail under the heading CHR$, and is used in many of the sample programs in this book.) For example, the reverse-video control function is coded 18 in ASCII; thus, the following statement would result in precisely the same output display as the example statement in Figure P.6:

```
PRINT CHR$(18); Q$; "VIC"; Q$
```

Another control character used often in this book is ASCII code 147, which clears the display screen and positions the cursor at the upper-left corner of the screen:

```
PRINT CHR$(147)
```

Which technique you use for activating control functions is a matter of personal preference. Those who prefer typing the control characters directly from the keyboard (between quotation marks) believe that this technique is faster and easier than using the CHR$ function. On the other hand, the argument in favor of the CHR$ function is that it makes the program listing itself easier to read and to document.

6. ***Printing numbers and the values of numeric variables.*** A variable name as an element of a PRINT statement will result in a screen display of the *value* of that variable. In this example the numeric variable N contains the value + 18, and the variable M contains the value -18. Notice that numeric value displays are automatically followed by a space; in addition, positive numeric values are also preceded by a space. You must always take this automatic spacing into account when you are planning output screens that contain many numeric values.

7. ***Arithmetic expressions and scientific notation.*** If an *arithmetic expression* is included in a PRINT statement, the computer will first evaluate that expression and then display the result on the screen. The resulting number will be displayed in *scientific notation* if it is less than .01; or greater than or equal to 10^9. (*See the entries under* Arithmetic Expression and Scientific Notation.)

8. ***Logical expressions.*** A logical expression in a PRINT statement results in 0 if the expression evaluates to false or -1 if the expression evaluates to true. (*See* IF.)

One final note: The PRINT statement alone, with no display elements, will result in a blank line on the screen. For example:

```
10 PRINT "SOMETHING"
20 PRINT
30 PRINT "SOMETHING ELSE"
```

Line 20 puts a blank line between the two lines of text.

Almost every sample program in this book contains examples of PRINT. Studying the illustrations in Figures P.5 and P.6 should help you understand these examples.

PRINT# (output command word)

The PRINT# command sends data to a file stored on cassette or disk. PRINT# must be preceded by an OPEN command that specifies the characteristics and location of the file. The format of the PRINT# command is as follows:

```
PRINT#F, V1; V2; V3; ...
```

where F is the file number specified in the OPEN statement, and V1, V2, and V3 represent a list of values or variables whose values will be sent to the file.

When sending data items to a file, you must always keep in mind how to distinguish between one data item and the next. When you read a file, using the INPUT# command, the items of the file must be separated by a character that INPUT# recognizes as a field marker: either the RETURN character or the comma. INPUT# reads either of these characters as a marker separating two data items. Therefore, after you send each data item to the file, you must make sure that one of these characters is also sent after the data item.

There are several techniques available for doing this. One simple way is to send each data item to the file via its own PRINT# statement. PRINT# automatically sends a RETURN character following the data it sends:

```
PRINT#1, V1
PRINT#1, V2
PRINT#1, V3
PRINT#1, V4
```

Each of these values, then, will be separated in the file by a RETURN character. Another method is to send a comma after each data item:

```
LET C$ = ","
PRINT#1, V1;C$;V2;C$;V3;C$;V4;C$
```

Each of these values will be separated by a comma; INPUT# will thus be able to read each value into a variable of its own.

Sample Program

The program shown in Figure P.7 illustrates the use of PRINT# for creating a sequential file of data. This program creates a file named EMPLOYEES, which contains records describing several employees in an imaginary company. For each employee, the program stores four items of

```
 10 OPEN 1,8,2,"EMPLOYEES,S,W"
 20 READ N
 25 PRINT#1, N
 30 FOR I= 1 TO N
 40 READ T$, L$, F$, S
 50 PRINT#1, T$
 60 PRINT#1, L$
 70 PRINT#1, F$
 80 PRINT#1, S
 90 NEXT I
100 CLOSE 1
110 END
120 DATA 8
130 DATA S,ALSTON,LOIS,1900
140 DATA H,DUFF,JOANNE,8.95
150 DATA H,GIBSON,DONALD,18.75
160 DATA S,INEZ,ROBERT,2600
170 DATA H,RACHEL,BEN,7.75
180 DATA H,SCULLY,LEE,21.29
190 DATA S,SHEPARD,CLARA,3000
200 DATA H,TIBBS,DANIEL,7.25
```

Figure P.7: PRINT#—Sample Program

data: a "tag"—either "S" or "H"—indicating the employee's status, either salaried or hourly; the employee's last name; the employee's first name; and the employee's wages—weekly if the tag is "S"; hourly if the tag is "H".

The program initially reads all this information from a series of DATA statements located at the end of the program listing itself. It stores the data items one by one in the file after reading each item from the DATA statements.

Line 10 of the program opens the sequential file for writing:

```
10 OPEN 1,8,2,"EMPLOYEES,S,W"
```

The first DATA item, and the first item to be sent to the file, is an integer that indicates how many employee records will be stored in the file. This data item will make the file easier to read later. (*See the entry under* INPUT#.):

```
20 READ N
30 PRINT#1, N
```

This value, N, the number of records, is stored in the first of the DATA statements:

```
120 DATA 8
```

After this item is stored, each employee record can be read from its DATA statement and sent to the file. This is done inside a FOR loop:

```
30 FOR I = 1 TO N
40    READ T$, L$, F$, S
50    PRINT#1, T$
60    PRINT#1, L$
70    PRINT#1, F$
80    PRINT#1, S
90 NEXT I
```

Notice that each data item is sent to the file by its own PRINT# statement; this ensures that each item will be followed by a RETURN character. When we read this file, the INPUT# statement will be able to distinguish between one item and the next.

When all the data is sent to the file, a CLOSE statement is required to close the file:

```
100 CLOSE 1
```

Program (computer vocabulary)

A program is a sequence of instructions, written in a computer language, designed to make the computer accomplish a specific task. The instruction lines of a BASIC program are numbered. The lines of a program may contain either a single instruction:

```
20 PRINT "CHECKBOOK BALANCE"
```

or multiple instructions, separated by colons:

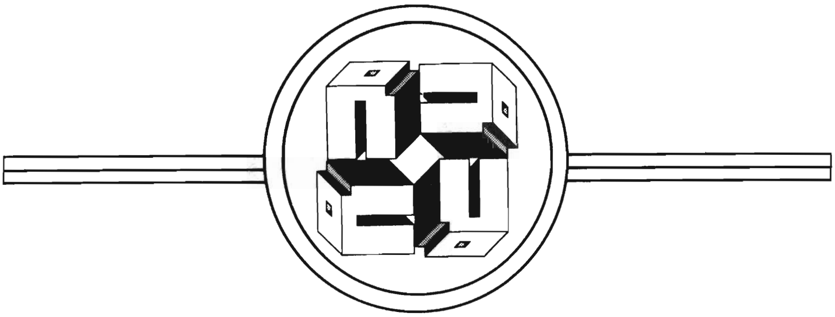
```
30 PRINT : PRINT V, B, D : GOTO 120
```

The computer merely holds the instructions of a program in its memory until you enter the RUN command, telling the computer to begin performing the program. (For this reason, writing instructions as numbered program lines is sometimes referred to as "deferred execution mode.")

A display of the lines of a program (either on the screen or on paper) is called a program "listing."

Programmer (computer vocabulary)

The programmer is the person who writes a computer program, as opposed to the "user," the person who runs, and often interacts with, a program. In the context of personal computers, however, the programmer and the user are often the same person.



READ (command word)

The READ command reads the data items stored in the program's DATA statements. READ accesses these data items sequentially, and assigns each item it reads to a *variable.*

A single READ statement may read one or more data values. The command word READ is followed by a list of variable names; the statement reads one value for each variable in the list. For example, consider the following statements:

```
10 READ V1
20 READ A, B, C, D
30 READ M$, N, I%
```

These statements read one, four, and three values, respectively. Notice that the values read by one statement need not all be of the same type; line 30, for example, reads a string value, a real numeric value, and an integer value.

In effect, the computer treats the values stored in DATA statements as a sequential file. A "pointer" to the current item in the file is automatically set up. Initially, this pointer is set to the first data value in the first DATA statement. After a READ statement accesses the current value, the pointer is moved forward to the next data value.

When you use the READ/DATA combination, you must keep track of the data type of each value that READ will access. If READ attempts to assign a string value to a numeric variable, your program will be terminated with a syntax *error message.* Likewise, in any program the total number of variables to be assigned values via READ statements must not exceed the number of data items available in DATA statements. If you try to

READ more values than exist, your program will terminate with the following *error message*:

```
? OUT OF DATA
  ERROR IN 10
```

where 10, in this instance, is the line number of the READ statement.

The entries under DIM and RND show programs that illustrate the READ and DATA statements. (*See also the entries under* DATA and RESTORE for more information.)

REM (command word)

The REM (for "remark") statement allows you to create short, but permanent, notes documenting your program. After the keyword REM, you may write any kind of comment or information that will help you remember and understand what your program does. For example:

```
10 REM ** THIS PROGRAM HELPS YOU BALANCE YOUR
          CHECKBOOK
```

REM statements result in no action; the computer simply ignores them during a program run. REM lines may appear anywhere in the program listing, not just at the beginning. Unfortunately, with limited memory, you have to be careful about the length and number of REM lines in any one program. (Even though they produce no action, they still take up memory space. *See the sample programs under* PEEK and POKE.) Particularly in long programs, you have to strike a compromise between the extra clarity supplied by REM commands and the need to conserve memory space.

For some significant examples of the use of REM, *see the sample program under the heading* GOSUB.

RESTORE (command word)

The RESTORE command is used in connection with the READ and DATA statements. The DATA statement allows you to store a "file" of data elements inside your BASIC program. The READ statement accesses these elements sequentially and assigns them to *variables*. The computer automatically sets up a "pointer" to identify the current data item in the DATA statement "file"; this pointer is initially set at the first value in the first DATA statement. Each time READ accesses a value, the pointer is incremented forward to the next data element.

Sometimes it is convenient to be able to reset this pointer back to the beginning of the "file," so that the data can be read an additional time. The RESTORE command performs this task. After RESTORE is performed, subsequent READ statements will begin reading values starting from the first DATA statement in the program.

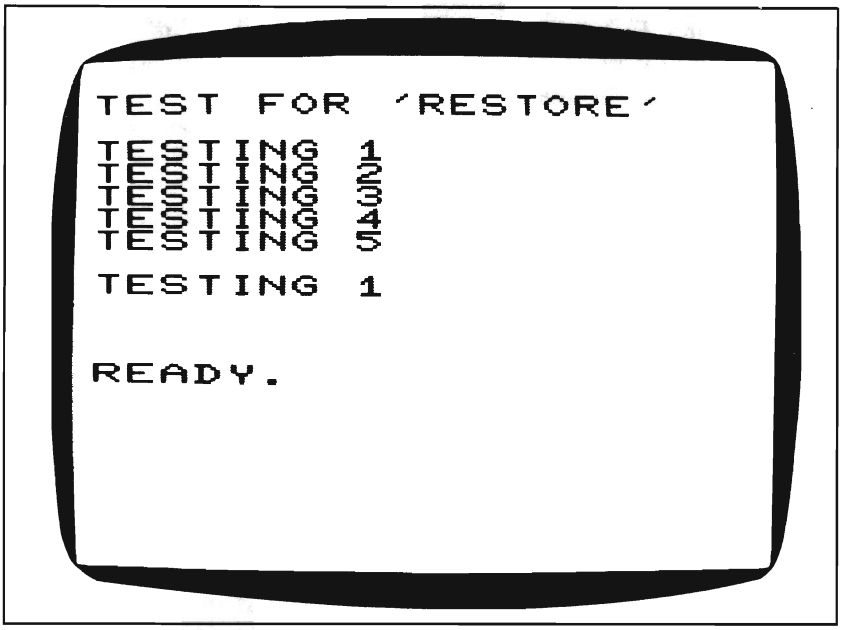
Sample Program

The short test program shown in Figure R.1 illustrates RESTORE in action. The program contains five DATA statements (lines 100 to 104), each of which contains two data items. These items are read for the first time, and displayed on the screen, by the FOR loop in lines 10 to 30. Line 60 shows the RESTORE command; after the first reading of the data, this line resets the pointer to the first data item in the set of DATA statements. To show that the pointer has indeed been reset, lines 70 and 80 read the first two data items again and display them on the screen. The output from the program appears in Figure R.2.

To see what would happen in this program if the RESTORE command were omitted, remove line 60 from the program and run the program a

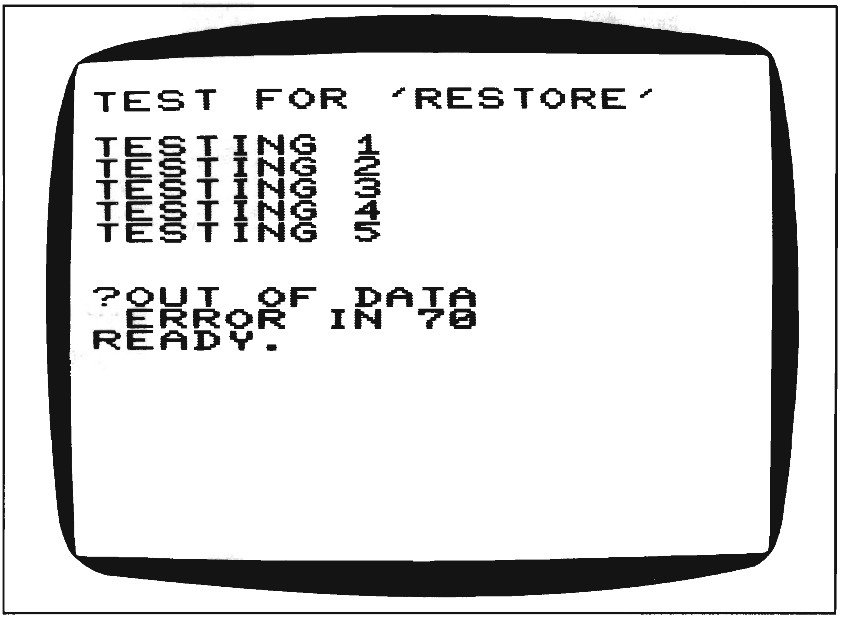
```
  5 PRINT CHR$(147)
  7 PRINT "TEST FOR 'RESTORE'"
  8 PRINT
 10 FOR I = 1 TO 5
 20   READ S$,N
 30   PRINT S$;N
 40 NEXT I
 50 PRINT
 60 RESTORE
 70 READ S$,N
 80 PRINT S$;N
 90 PRINT:PRINT:END
100 DATA TESTING,1
110 DATA TESTING,2
120 DATA TESTING,3
130 DATA TESTING,4
140 DATA TESTING,5
```

Figure R.1: RESTORE—Sample Program



```
TEST FOR 'RESTORE'
TESTING 1
TESTING 2
TESTING 3
TESTING 4
TESTING 5
TESTING 1


READY.
```

Figure R.2: RESTORE—Sample Output



```
TEST FOR 'RESTORE'
TESTING 1
TESTING 2
TESTING 3
TESTING 4
TESTING 5

?OUT OF DATA
 ERROR IN 70
READY.
```

Figure R.3: Error Message without RESTORE

second time. (To remove a program line, simply type the line number and then press the RETURN key.) Figure R.3 shows the results. When line 70 tries to read another data item, there are no more items to be read (because the pointer is at the end of the list of items), and the program ends with the following *error message*:

```
? OUT OF DATA
  ERROR IN 70
```

RETURN (command word)

The RETURN statement marks the end of a *subroutine*. RETURN tells the computer to send control of the program back to the line immediately following the GOSUB statement that originally called the subroutine. For a detailed description and examples, *see the entry under* GOSUB.

RIGHT$ (string function)

The RIGHT$ *function* allows you to access the last *n* characters of a *string*. The function takes the form:

```
RIGHT$(S$,N)
```

where S$ may be expressed as a *literal* string, a string *variable*, or a string expression (i.e., a *concatenation*), and N represents an integer. The function returns the last N characters of S$; for example:

```
PRINT RIGHT$("HANDBOOK",4)
```

This statement would display the word BOOK on the screen.

Notes and Comments

— *See also* LEFT$ and MID$, two other string functions that allow you to access a portion of a string. The sample program under the heading TI$ shows all three of these functions in action.

RND (function)

Each call to the RND *function* returns a random number. Actually, the numbers that RND generates are the result of a complex calculation that the computer performs. This means that they are not truly random; but

they are certainly random-seeming enough for most programs.
The random numbers, R, produced by RND are in the range:

$0 <= R < 1$

The *argument* of the RND function controls the "seed" of the function, determining the starting point of a series of random numbers produced by RND. The format of a call to RND is:

```
RND(S)
```

where S is any real number. The value of S determines how the function will operate:

1. If S is any positive number, successive calls to RND will produce an unpredictable series of random numbers.
2. If S equals zero, RND will also produce random-seeming numbers, but the computer will use a different method for calculating these numbers. The calculation, in this case, is based on the current value of TI, the computer's clock function.
3. For a given negative value of S, RND always returns the same "random" number. Furthermore, you can produce two identical series of random numbers if you begin each series by calling RND with the same negative argument. For example, the following program will always produce the same series of random numbers, no matter when or how often you run it:

```
10 PRINT RND(-1)
20 FOR I = 1 TO 5
30     PRINT RND(1)
40 NEXT I
```

The call to RND in line 10, with a negative argument, determines a fixed, predictable starting point for the series of numbers. As a result, the call to RND inside the FOR loop (line 30; a positive argument) always produces the same five random numbers. If you delete line 10 from the program, however, the starting point of the series will be unpredictable, and the FOR loop will produce a new and different set of random numbers each time it is performed. (Type these lines into your computer and perform this experiment for yourself.)

In some programming situations you may want the computer to produce the same set of random numbers time after time. For example, you may wish to re-examine the play of a game that depends on random numbers; or you might want to duplicate a certain scenario of a simulation model. Both of these examples might involve running a program a number of times and being certain that the computer will generate the same sequence of random numbers for each run. You can accomplish this by including a line in your program similar to line 10 above. Then, when you are ready to start running the program on unpredictable series of random numbers, you can simply delete the line.

The range of numbers returned by RND—between 0 and 1—is not very convenient for many applications. The following formula is commonly used to convert the range; it supplies random integers, R, between L (for "low") and H (for "high"), inclusive:

```
LET R = INT((H - L + 1) * RND(1)) + L
```

For random integers, R, in the range from 1 to H, inclusive, the formula reduces to:

```
LET R = INT(H * RND(1)) + 1
```

In this latter formula, the expression:

```
H * RND(1)
```

produces a number between 0 and H. Taking the integral value of this expression:

```
INT(H * RND(1))
```

yields an integer from 0 to H-1. Finally, adding a value of 1 gives the desired range: 1 to H.

Note that an argument of zero for RND may also be used in this formula.

Sample Program

The program in Figure R.4 shows several of the *algorithms* required for a computerized card game—in particular, an efficient algorithm for "shuffling" the cards, using the RND function. The output from this program is simply a list of the 52 cards in their shuffled order. Figure R.5 shows the output screen.

The shuffling program stores the deck in the array D. Line 10 defines the array:

```
10 DIM D(52)
```

```
  5 REM ** CARD SHUFFLER
 10 DIM D(52)
 15 GOSUB 200
 20 GOSUB 450
 30 GOSUB 300
 40 GOSUB 500
 45 GOSUB 500
 48 REM **DISPLAY CARDS
 50 FOR I = 1 TO 13
 60   FOR J = 0 TO 3
 70     LET C=D(I+J*13)
 80     LET S=INT((C-1)/13)+1
 90     LET R=C-13*(S-1)
100     PRINT TAB(J*5+2);R$(R);S$(S);
110   NEXT J
120   PRINT
130 NEXT I
140 PRINT:PRINT
150 END
200 PRINT CHR$(147)
210 PRINT "    SHUFFLED DECK"
220 PRINT "       52  CARDS"
230 PRINT
240 RETURN
300 REM ** SUITS AND  RANKS
310 DIM S$(4),R$(13)
320 FOR I = 1 TO 13
330   READ R$(I)
340 NEXT I
350 FOR I = 1 TO 4
355   READ C
360   LET S$(I)=CHR$(C)
370 NEXT I
380 DATA A,2,3,4,5
390 DATA 6,7,8,9,10
400 DATA J,Q,K
```

Figure R.4: RND—Sample Program

```
410 DATA 211, 218, 216 , 193
420 RETURN
450 REM ** CREATE THE DECK
460 FOR I = 1 TO 52
470   LET D(I) = I
480 NEXT I
490 RETURN
500 REM ** SHUFFLE
510 FOR I = 1 TO 52
520   LET R = INT(RND(0)*52)+1
530   LET H = D(R)
540   LET D(R)=D(I)
550   LET D(I)=H
560 NEXT I
570 RETURN
```

Figure R.4: RND—Sample Program, continued

Each card is represented in this array by an integer from 1 to 52. Initially, you can think of the cards as arranged in order of rank, suit by suit. The program shuffles the deck by rearranging the integers stored in D in a random order. Once this shuffling process is complete, the program uses other algorithms to translate each integer into the name of a card. We'll see how all this is done as we examine the program's major subroutines; here is a brief summary of those routines:

1. ***Create the deck*** (subroutine at line 450). Initializes the array D.
2. ***Shuffle the cards*** (subroutine at line 500). Rearranges the order of the values in D.
3. ***Initialize the card names*** (subroutine at line 300). Creates two string arrays: S$ for the graphics symbols that represent the suits; and R$ for the names of the ranks.
4. ***Determine the name of each card*** (main program section, lines 50 to 130). Translates each integer from 1 to 52 into a suit symbol (from S$) and a rank name (from R$).

To create the deck, the subroutine at line 450 simply assigns the integers 1 to 52, in order, to the 52 elements of D:

```
460 FOR I = 1 TO 52
470   LET D(I) = I
480 NEXT I
```

To rearrange these 52 integers, the shuffling subroutine chooses, at random, a new position in D for each card of the deck. This is performed inside a FOR loop:

```
510 FOR I = 1 TO 52
```

The first line inside the loop shows an example of the RND function in action:

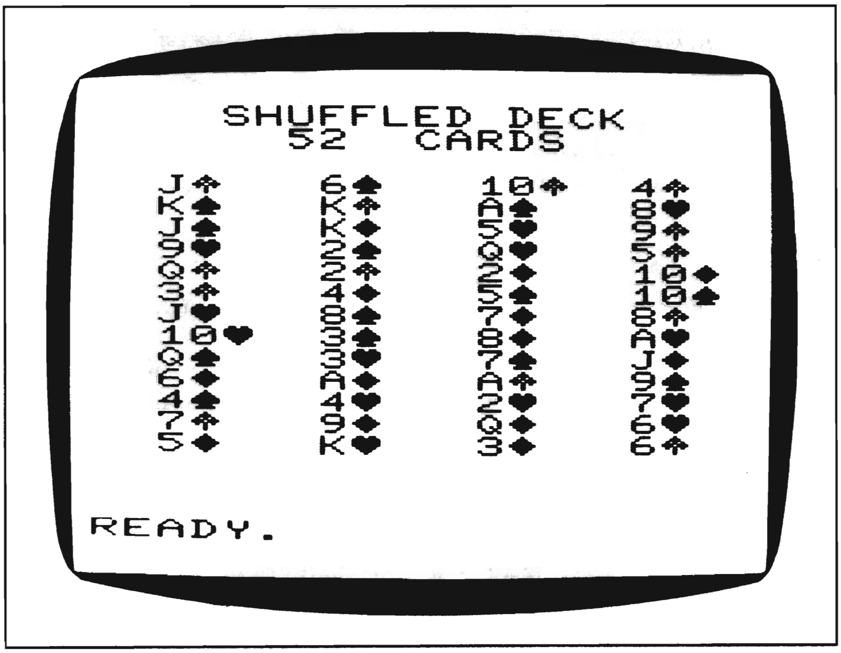
```
520 LET R = INT(RND(0) * 52) + 1
```

This line chooses a random number from 1 to 52 and stores the number in the variable R. The next three lines swap the cards in the deck positions represented by D(I) and D(R). (This swapping operation is reminiscent of a sorting algorithm. *See the entry under* Algorithm.) First, the value of D(R) is saved in a holding variable, H:

```
530 LET H = D(R)
```

Then the randomly chosen array element D(R) receives a new value, the value of D(I):

```
540 LET D(R) = D(I)
```



```
        SHUFFLED DECK
          52   CARDS

  J♣       6♠       10♣      4♣
  K♠       K♣       A♠       8♥
  J♠       K♦       5♥       9♣
  9♥       2♠       Q♥       5♣
  Q♣       2♣       2♦       10♦
  3♣       4♦       5♠       10♠
  J♥       8♠       7♦       8♣
  10♥      3♠       8♦       A♥
  Q♠       3♥       7♠       J♦
  6♦       A♦       A♣       9♠
  4♠       4♥       2♥       7♥
  7♣       9♦       Q♦       6♥
  5♦       K♥       3♦       6♣

READY.
```

Figure R.5: RND—Sample Output

And finally D(I) receives the original value of D(R), now stored in H:

```
550 LET D(I) = H
```

As the FOR loop increments the control variable I from 1 to 52, each card in the deck is swapped with a randomly chosen card elsewhere in the deck. By the time I reaches 52, the deck is completely shuffled.

The subroutine at line 300 determines the original order of the deck in the way it assigns the card names to the arrays S$ and R$. These arrays are initialized via two READ statements (lines 330 and 355), which read the data stored in lines 380 to 410. The elements of R$—from R$(1) to R$(13)—store the rank names, A, 2, 3, ..., Q, K. The elements of S$—from S$(1) to S$(4)—store the suit symbols: the heart, the diamond, the club, and the spade. These graphics characters are displayed on the keyboard below the letters S, Z, X, and A, respectively. The subroutine reads the ASCII codes for these four characters from the last DATA statement, and then uses the CHR$ function to assign the characters themselves to the array R$:

```
355 READ C
360 LET S$(I) = CHR$(C)
    ...
410 DATA 211, 218, 216, 193
```

As a result of this subroutine, the suits are initially arranged as follows:

cards 1 to 13: hearts
cards 14 to 26: diamonds
cards 27 to 39: clubs
cards 40 to 52: spades

Within each suit the first card is the ace, the second through tenth are the number cards, and the last three are the face cards.

The main program section, lines 50 to 130, has the somewhat complex job of determining the name of each card, given a card number from 1 to 52. To do so it must reduce each card number, C, into two numbers:

S = the suit number (from 1 to 4)
R = the rank number (from 1 to 13)

The relationship among these numbers is expressed in the formula:

$$C = 13 \times (S - 1) + R$$

The arithmetic required to find S and R for each card is performed in lines

80 and 90. Once these values are determined, they can be used as indexes into the string arrays S$ and R$, to display the name of a card on the screen:

```
100 PRINT TAB(J*5+2); R$(R); S$(S);
```

After shuffling the deck twice (lines 40 and 45), the main program section arranges to display the entire shuffled deck in four columns on the screen. The nested FOR loops starting at lines 50 and 60 perform this task.

RUN (command word)

RUN instructs the computer to begin performing the current program in memory. If you enter the command simply as:

```
RUN
```

the performance will begin with the first line of the program. If you enter the command as:

```
RUN N
```

where N is a *literal* numeric *value* representing a line number, the performance will begin at line N. In both cases, the computer first clears out of its memory the values of any *variables* left over from previous program runs, and then begins executing the program. (*See* GOTO.)

SAVE (program storage command)

The SAVE command creates or overwrites a BASIC program *file*, saving the current program onto a cassette or disk. The command takes several formats; perhaps the most general and common format is:

```
SAVE "PROGRAM NAME", D
```

where the program name, entered between quotation marks (or as a *string variable*), may be from 1 to 16 characters long; and the D parameter is a value representing the device number. For a cassette, the device number is 1; for a disk, the device is 8.

When you are storing a program on cassette, you may omit the device number from the command; the "default" device number is 1:

```
SAVE "PROGRAM NAME"
```

If you have not turned the cassette recorder on at this point, the computer will display the message:

```
PRESS RECORD & PLAY ON TAPE
```

When you turn on the recorder, the computer will save the program and automatically stop the operation of the recorder when the saving process is complete. You may also save programs on tape without specifying a program name, but in general it is a better idea to give each program an identifying name.

For saving a program on disk, both the name and the device number are required parameters of the SAVE command:

```
SAVE "PROGRAM NAME", 8
```

As long as no program already exists on disk under the specified name, this command will create a new program file and successfully save the program. If a program does exist under this name, and you wish to overwrite this program file, replacing it with the current version of the program, then you must use the following command:

```
SAVE "@:PROGRAM NAME", 8
```

The @ character tells the computer that this is an overwrite operation. For example, let's say you have written a program named TAXES, and saved it on disk under that name. If you subsequently revise the program, and you wish to save the new version under the same name, you would give the command:

```
SAVE "@:TAXES", 8
```

The original version of TAXES will be lost, replaced by the new version that you have just saved. If you want to save both versions, you will have to save the new version under a different name; for example:

```
SAVE "TAXES2", 8
```

Notes and Comments

— The SAVE command has an additional parameter, for use with cassette storage:

```
SAVE "PROGRAM NAME", 1, S
```

If S equals 1, subsequent LOAD commands will load the program into the same memory location from which it was saved. If S equals 2, an end-of-tape marker will be added to the end of the program file. If S equals 3, both of these features will be activated.

Scientific Notation (computer vocabulary)

Scientific notation is a system of writing numbers in two distinct components: the mantissa (the significant digits of the number) and the exponent (the power of 10 that indicates the location of the decimal point in the number). Your computer uses scientific notation to display very small or very large numbers on the screen. For example:

```
2.34 E + 14
```

In this number, the value located before the letter E is the mantissa, and the value located after E is the exponent of 10. You can read this number as "2.34 times 10 to the 14th power":

2.34 × 100,000,000,000,000

or:

234,000,000,000,000

A negative exponent translates into a fractional value. For example:

```
8.7 E-10
```

means:

.00000000087

SGN (function)

The SGN *function* identifies the sign of any number. SGN takes the form:

```
SGN(N)
```

where N is a *literal* numeric *value*, a numeric *variable*, or an *arithmetic expression*. It returns one of the following values:

-1	if $N < 0$
0	if $N = 0$
+1	if $N > 0$

Sample Program

The SGN function can be useful whenever a program defines different courses of action, the choice among which depends on the sign of a number. The BASIC program shown in Figure S.1 is an exercise illustrating the use of SGN in such a situation. Three subroutines are set aside at lines 100, 200, and 300, for the cases $N < 0$, $N = 0$, and $N > 0$, respectively. The subroutine call is in line 60:

```
60 ON SGN(N) + 2 GOSUB 100, 200, 300
```

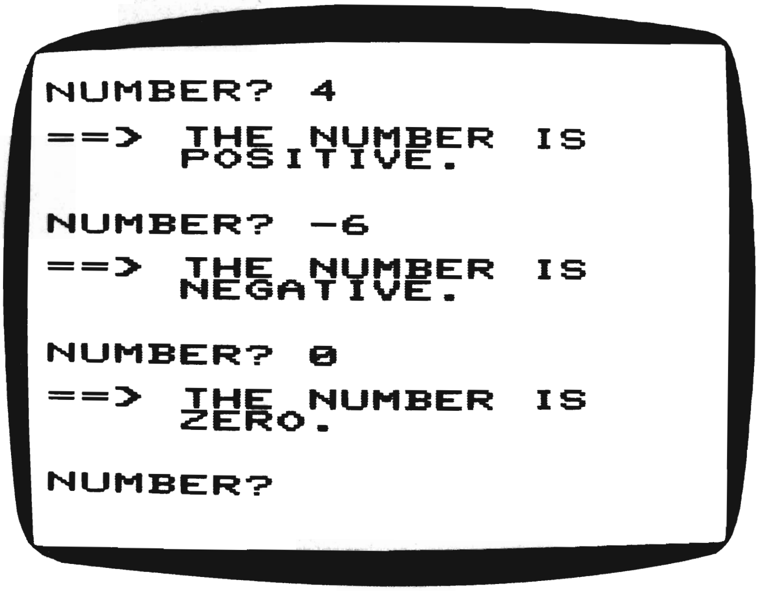
Since SGN(N) results in an integer from -1 to +1, the expression:

```
SGN(N) + 2
```

gives an integer from 1 to 3, resulting in a call to one of the three subroutines listed after the word GOSUB. (*See* GOSUB and ON.)

```
  5 PRINT CHR$(147)
 10 REM ** SIGN OF N
 20 REM ** DETERMINES
 25 REM ** THE ACTION.
 30 REM
 40 INPUT "NUMBER"; N
 45 PRINT
 50 PRINT "==> THE NUMBER IS "
 55 PRINT "    ";
 60 ON SGN(N)+2 GOSUB 100,200,300
 70 PRINT:PRINT
 80 GOTO 40
100 PRINT "NEGATIVE."
110 RETURN
200 PRINT "ZERO."
210 RETURN
300 PRINT "POSITIVE."
310 RETURN
```

Figure S.1: SGN—Sample Program



```
NUMBER? 4
==> THE NUMBER IS
    POSITIVE.

NUMBER? -6
==> THE NUMBER IS
    NEGATIVE.

NUMBER? 0
==> THE NUMBER IS
    ZERO.

NUMBER?
```

Figure S.2: SGN—Sample Output

Without the benefit of the SGN function, the program would require three lines to decide which subroutine to call:

```
55 IF N < 0 THEN GOSUB 100
60 IF N = 0 THEN GOSUB 200
65 IF N > 0 THEN GOSUB 300
```

Figure S.2 shows a sample run of this program.

SIN (function)

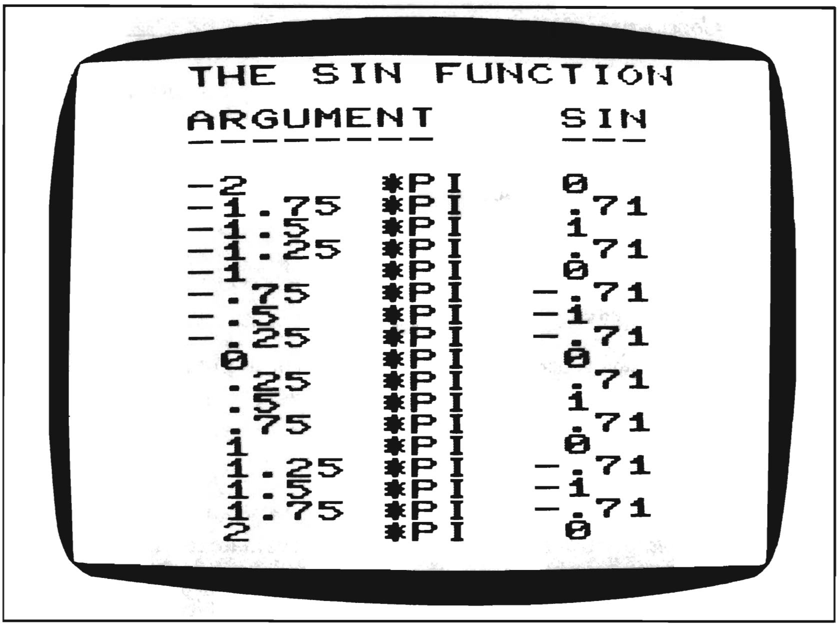
Given any angle (negative or positive) expressed in radians, the SIN *function* returns the sine of the angle.

Sample Program

The program shown in Figure S.3 displays a series of sine values for *arguments* ranging from -2π to $+2\pi$. The output from this program appears in Figure S.4.

```
  5 DEF FNR(X)= INT(100*X+.5)/100
 10 PRINT CHR$(147)
 20 PRINT TAB(3); "THE SIN FUNCTION"
 30 PRINT
 35 PRINT "    ";
 40 PRINT "ARGUMENT          SIN"
 45 PRINT "    ";
 50 PRINT "-------           ---"
 60 PRINT
 70 FOR I = -2 TO 2 STEP 1/4
 75 PRINT "    ";
 80 PRINT I; TAB(9);"*PI";TAB(14); FN R(SIN(I*3.1416))
 90 NEXT I
100 GET A$
110 IF A$="" GOTO 100
```

Figure S.3: SIN—Sample Program



```
THE SIN FUNCTION
ARGUMENT       SIN
--------       ---

-2      *PI    0
-1.75   *PI    .71
-1.5    *PI    1
-1.25   *PI    .71
-1      *PI    0
-.75    *PI   -.71
-.5     *PI   -1
-.25    *PI   -.71
 0      *PI    0
 .25    *PI    .71
 .5     *PI    1
 .75    *PI    .71
 1      *PI    0
 1.25   *PI   -.71
 1.5    *PI   -1
 1.75   *PI   -.71
 2      *PI    0
```

Figure S.4: SIN—Sample Output

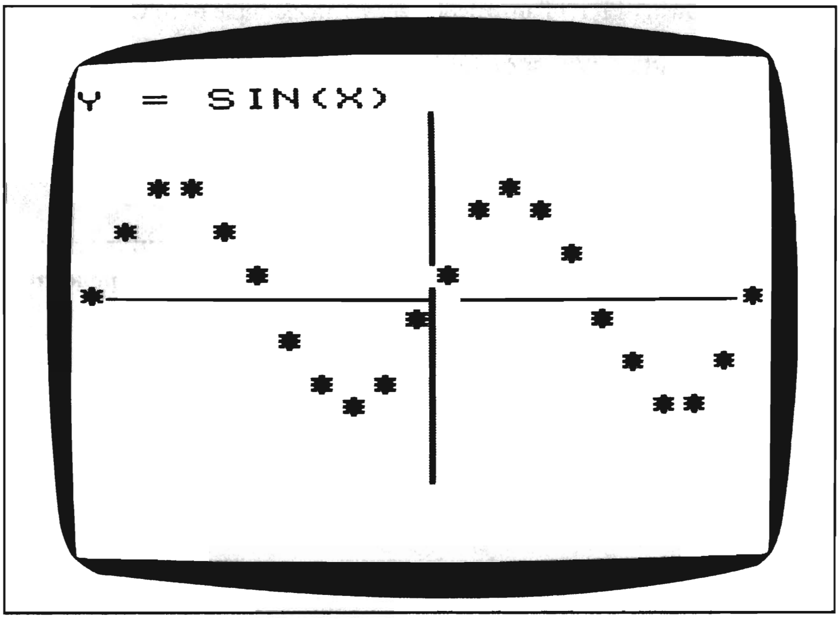


Figure S.5: SIN—Plotted Graph

Notes and Comments

- Figure S.5 shows a crude graph of the sine function from -2π to + 2π, produced on the VIC-20 computer.
- *See the entries under* COS and TAN for more information about the trigonometric functions.

SPC (screen display function)

The SPC *function* is used in a PRINT statement to put a specified number of space characters in the current output line. The format of SPC is:

```
SPC(N)
```

where N is the number of spaces. N may be expressed as a literal numeric value, a *variable*, or an *arithmetic expression*; it must evaluate to a number in the following range:

$0 <= N <= 255$

The following PRINT instruction shows an example of SPC:

```
PRINT "X"; SPC(15); "Y"
```

In the output line, there will be 15 spaces between X and Y.

SQR (function)

Given a nonnegative numeric *argument*, the SQR *function* supplies the square root.

Sample Program

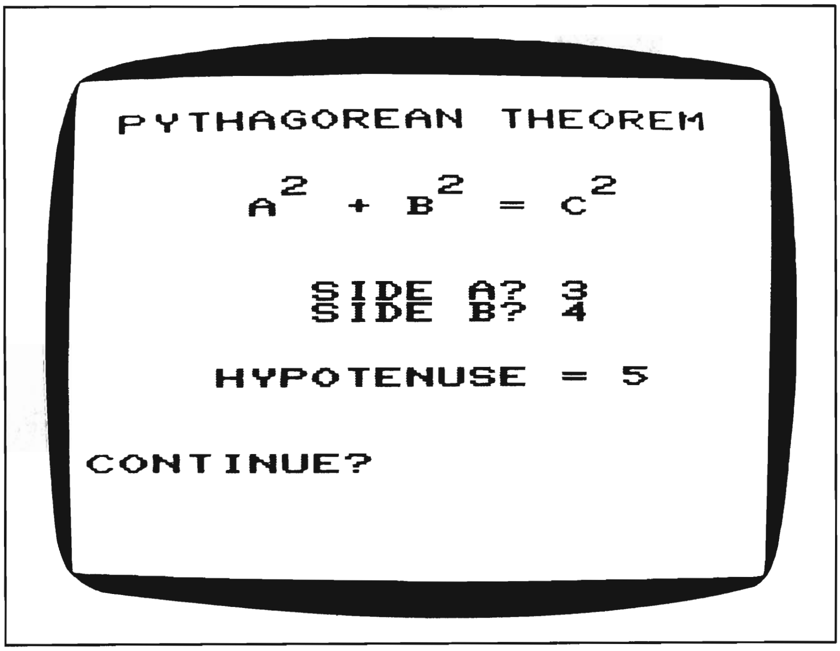
Figure S.6 shows a program that calculates the length of the hypotenuse of a right triangle, given the lengths of the two sides, represented by A and B. Program line 90 finds the hypotenuse, C, using the SQR function:

```
90 LET C = SQR(A*A + B*B)
```

A sample of this program's screen output appears in Figure S.7.

```
  5 PRINT CHR$(147)
 10 PRINT " PYTHAGOREAN THEOREM"
 20 PRINT:PRINT
 30 PRINT "          2     2     2"
 40 PRINT "        A   + B   = C "
 50 PRINT:PRINT:PRINT
 60 INPUT "          SIDE A";A
 70 INPUT "          SIDE B";B
 80 PRINT:PRINT
 90 LET C = SQR(A*A + B*B)
100 PRINT "     HYPOTENUSE =";C
110 PRINT:PRINT:PRINT
120 INPUT "CONTINUE";A$
130 GOTO 5
```

Figure S.6: SQR—Sample Program



```
 PYTHAGOREAN THEOREM

        2     2     2
       A  + B  =  C


          SIDE A? 3
          SIDE B? 4

     HYPOTENUSE = 5


CONTINUE?
```

Figure S.7: SQR—Sample Output

ST (input and output function)

The ST *function* supplies a number indicating the status of an input or output operation. Specifically, the value of ST gives you information about an open file after an input or output operation. The values of ST that you will probably find most significant, and the most useful to check for, are:

0	— no error
64	— end of file

Other values indicate I/O errors of various sorts.

The sample program under the heading GET# shows an illustration of the use of ST. The program reads a sequential file, character by character, and requires some way of recognizing the end of the file. The following test is performed after each GET# operation:

```
30 IF ST = 64 GOTO 60
```

If the previous GET# found the end of the file, ST will report a value of 64. In that case, control of the program is sent to line 60, which closes the file and leads to the end of the program:

```
60 CLOSE 1
70 PRINT : PRINT : END
```

Without the test in line 30, the program would continue reading characters from other files, producing garbage results. Note that the ST report in line 30 will work both for cassette and disk data files.

STEP (command word)

The STEP clause in a FOR statement indicates how much the control *variable* will be incremented (or decremented) for each pass through the loop. STEP is optional in a FOR statement; without it, the "default" incrementation value is 1.

For example, the following FOR statement introduces a loop that has the control variable I:

```
FOR I = 0 TO 25
```

In this case the loop will go through 26 iterations, with I taking the values 0, 1, 2, 3, ..., 25. Adding a STEP clause to this statement changes both the series of values that I will take, and the number of iterations that the loop will go through:

```
FOR I = 0 TO 25 STEP 5
```

Now the loop will repeat only 6 times, with I taking the values 0, 5, 10, 15, 20, and 25.

The STEP clause can also specify negative and fractional incrementation amounts. For example, the following FOR statement defines a decrementing control variable:

```
FOR I = 20 TO 0 STEP -2
```

Notice first that the number before TO is greater than the number after TO; if not for the STEP clause, this FOR loop would perform only a single iteration. STEP, however, defines the decrementation amount as -2. The control variable will thus take values from 20 down to 0; i.e., 20, 18, 16, 14, ..., 0.

Finally, in the following loop the range of the control variable I will be from -1 to + 1 in increments of .1:

```
FOR I = -1 TO 1 STEP .1
```

I will take the values -1, -.9, -.8, ..., 0, .1, .2, ..., 1.

Sample Program

Figure S.8 offers a program, just for fun, in which one of the computer's bugs comes out to demonstrate the STEP clause. This is a simple example of a moving graphics program. When you run it you'll see a fairly innocuous-looking bug start moving diagonally up and down the screen. Figure S.9 shows the bug, but not the movement. You'll have to run the program for the full effect. (Note that this version of the program is designed for the VIC-20 computer. Because the program relies on the TAB *function* to position the bug on the screen, a few of the lines will have to be revised if you want to run the program on the Commodore 64 computer. See below.)

This program makes use of several of the Commodore keyboard graphics characters. The *subroutine* at line 300 creates the bug by storing sequences of these characters in elements of the string array B$. The characters themselves are represented in their respective ASCII codes in the DATA statements from line 370 to line 410. The subroutine reads each code number, and *concatenates* the character equivalent onto the appropriate string element of B$, thus building the bug shape, character by character:

```
330 READ C
340 LET B$(I) = B$(I) + CHR$(C)
```

Alternatively, we could have simply typed the graphics characters directly into the program from the keyboard, assigning a string of characters to each element of B$. (*See the entry under* CHR$.)

The FOR loop at lines 60 to 140 creates the moving bug. The FOR statement introduces the control variable I:

```
60 FOR I = F TO L STEP S
```

Inside the loop, the variable I will serve as an argument of TAB, thus determining where on the screen the bug will be placed. All the values in the FOR statement are represented by variables. The range variables, F (for "first") and L (for "last"), and the STEP variable, S, are initialized in lines 30 to 50. These initial values give the FOR loop the following effect:

```
60 FOR I = 1 TO 254 STEP 23
```

At this point in the program, then, the control variable I will take the values

```
  5 PRINT CHR$(147)
 10 REM ** JUMPING BUG
 20 GOSUB 300
 30 LET F=1
 40 LET L=254
 50 LET S=23
 60 FOR I=F TO L STEP S
 70   LET T = I-INT(I/22)*22
 80   PRINT TAB(I);B$(1)
 90   FOR J=2 TO 5
100     PRINT TAB(T);B$(J)
110   NEXT J
120   GOSUB 200
130   PRINT CHR$(147)
140 NEXT I
150 LET H=F
160 LET F=L
170 LET L=H
```

Figure S.8: STEP—Sample Program

```
180 LET S=-S
190 GOTO 60
200 REM ** PAUSE
210 FOR K=1 TO 100
220 NEXT K
230 RETURN
300 REM ** INIT BUG
310 FOR I = 1 TO 5
320   FOR J = 1 TO 6
330     READ C
340     LET B$(I)=B$(I)+CHR$(C)
350   NEXT J
360 NEXT I
370 DATA 202,213,192,192,201,203
380 DATA 32,221,46,46,221,32
390 DATA 32,221,213,201,221,32
400 DATA 32,202,192,192,203,32
410 DATA 32,206,32,32,205,32
420 RETURN
```

Figure S.8: STEP—Sample Program, continued

1, 24, 47, 70, 93, 116, 139, 162, 185, 208, 231, and 254 during the twelve iterations of the loop. Since the TAB function uses I to determine the location of each bug, these values send the bug down the screen:

80 PRINT TAB(I); B$(1)

The other string elements of B$ are printed at TAB(T) (lines 90 to 110); T is calculated, in line 70, as I *modulus* 22. Each time a new bug is displayed on the screen, the subroutine at line 200 is called. This subroutine simply creates a short pause in the action. Following the pause, the screen is cleared and the next bug is displayed at a new position on the screen. This sequence of events creates the illusion of movement.

After the first complete performance of the FOR loop (when the bug has reached the bottom of the screen), lines 150 to 170 swap the values of F and L, and line 180 reverses the sign of S. The GOTO statement in line 190 then sends control back up to the beginning of the FOR loop. With the new values of F, L, and S, the effect of the FOR loop will be:

50 FOR I = 254 TO 1 STEP -23

and I will take the values 254, 231, 208, 185, and so on, down to 1, sending

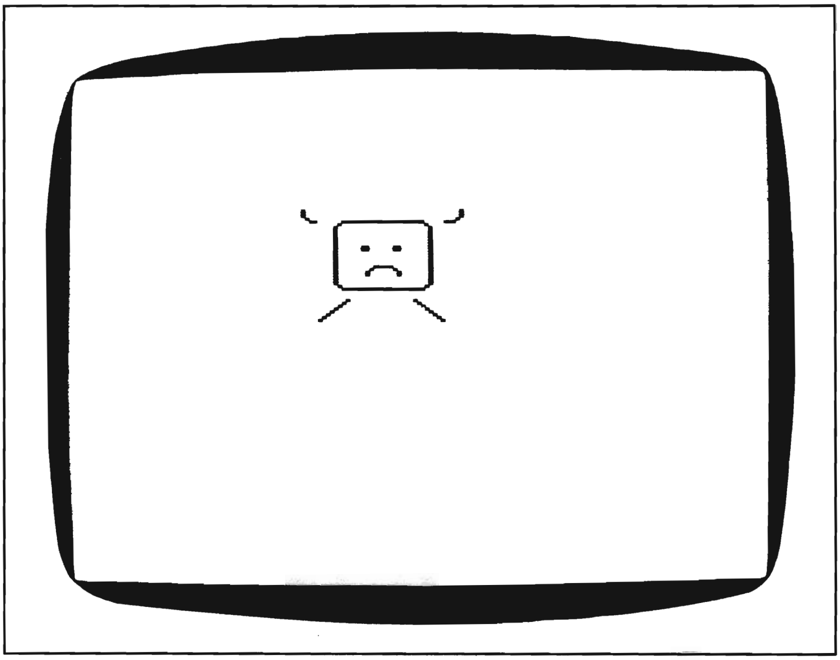


Figure S.9: STEP—Sample Output

the bug back up the screen. This process continues until you press the STOP key to stop the program run. Each time the FOR loop completes its action, the instructions in lines 150 to 180 reverse the values of the variables, and the GOTO command in line 190 starts the process over again.

To run this program on the Commodore 64 computer, revise lines 40, 50, and 70 as follows:

```
40  LET L = 252
50  LET S = 42
    ...
70  LET T = I - INT(I/40) * 40
```

STOP (command word)

The STOP command tells the computer to stop execution of a program run. When the computer encounters STOP, the program will terminate with a message such as:

```
BREAK IN 300
```

where 300, in this instance, is the line number of the STOP command.

The END statement may also be used to halt a program, but END produces no termination message. (*See the entry under* END.)

Sample Program

The program in Figure S.10 illustrates a situation in which either STOP or END is essential for the correct flow of program control. Normally, when the computer performs a BASIC program that contains no STOP or END command, it simply performs each line from beginning to end, until there are no more lines to perform. In that case the STOP command is not necessary; the computer stops on its own when the program is finished.

In a program that contains *subroutines*, however, the situation is different. Usually the subroutines are located at the end of the program listing; the top section of the program, sometimes called the "main program" section, calls the subroutines at the appropriate moments.

The program in Figure S.10 is simply an outline of this kind of program structure. Lines 10 to 50 form the main program, and the subroutines are at lines 100, 200, and 300. After all the subroutines are called, the computer must be told explicitly to stop, or else control of the program will simply continue down into the subroutine instructions.

If you run this program, which performs no real action, you will see the message:

```
BREAK IN 50
```

when the program run is complete. This tells you that the computer did

```
 10 REM ** MAIN PROGRAM
 20 GOSUB 100
 30 GOSUB 200
 40 GOSUB 300
 50 STOP
100 REM * SUBROUTINE 1
110 RETURN
200 REM * SUBROUTINE 2
210 RETURN
300 REM * SUBROUTINE 3
310 RETURN
```

Figure S.10: STOP—Sample Program

indeed stop at line 50. To see what would happen without the STOP command, try removing line 50 and running the program again. You will get the error message:

```
? RETURN WITHOUT GOSUB
ERROR IN 110
```

This means that the computer encountered a RETURN statement, at line 110, that was not preceded by a GOSUB command. Control of the program moved, improperly, into the subroutine instructions.

String (computer vocabulary)

A string is a nonnumeric data item that consists of one or more characters. The computer stores a string one character to a byte of memory, with each character translated into its numeric code equivalent. (*See the entries under* ASC, CHR$, STR$, and VAL.) BASIC offers several *functions* that take string *arguments* and return string values, including MID$, LEFT$, and RIGHT$, which are convenient functions for accessing a portion of a string. In addition, the function LEN returns the length, in characters, of a string.

STR$ (function)

Given a numeric *argument*, the STR$ *function* returns a *string* version of the numeric value. Specifically, the string returned by STR$ consists of the characters the computer would put on the screen to display the number. (In the case of positive numbers, the string returned by STR$ includes one leading space character. See the entry under PRINT.)

For example, consider the following lines:

```
LET N = 157.321
LET A$ = STR$(N)
```

The second line will store the string value "157.321" in the string variable A$. The difference between the value of N and the value of A$ lies in the way the computer stores numeric and string values. Real numeric values, such as the value of N, are stored in a "floating-point" system; the significant digits of the number (the "mantissa") and the exponent of the number (i.e., the power of 10 that indicates the location of the number's decimal point) are stored as two separate entities, for convenience and for maximum accuracy given a very large range of numbers. String values, such as the value of A$, on the other hand, are stored character by character, in their ASCII character code equivalents, one character to a byte of memory. (*See the entries under* CHR$ and ASC for an explanation of this code.)

Sample Program

Converting a number to a string allows you to manipulate the string for display purposes. The program listed in Figure S.11 converts a numeric input value representing dollars and cents into a numeric string display that includes a dollar sign, commas, and an alignable decimal point. Figure S.12 shows some sample output from this program. Some BASICs have a PRINT USING command, which performs a task similar to this, but the Commodore version of BASIC does not.

The dollar-and-cent formatting is performed by the subroutine at line 200. This subroutine is long and complicated, mostly because it must allow for several different kinds of input values. The main program section (lines 5 to 70) reads input values into the variable N. The subroutine ultimately stores each formatted dollar-and-cent string in the variable P$.

The subroutine begins by rounding the number N to the nearest cent, and then converting it to a string, which is assigned to the variable N$:

```
200 LET N = INT(N * 100 + .5)/100
210 LET N$ = STR$(N)
```

Line 215 then eliminates the leading space of the number string. The MID$ function accesses the second through last characters of N$, which are then reassigned to N$ itself:

```
215 LET N$ = MID$(N$,2,LEN(N$)-1)
```

```
 5 PRINT CHR$(147)
10 PRINT "TEST PROGRAM FOR"
15 PRINT "DOLLAR AND CENT FORMAT"
20 PRINT "INPUT VALUES:"
25 PRINT
30 INPUT N
40 GOSUB 200
45 PRINT
50 PRINT "  ==> ";P$
```

Figure S.11: STR$—Sample Program

```
 60 PRINT
 70 GOTO 30
140 REM * DOLLAR AND
150 REM * CENT FORMAT
160 REM * ROUTINE.
170 REM * RECEIVES
175 REM * VALUE IN N;
180 REM * RETURNS
190 REM * PRINT STRING
195 REM * IN P$.
200 LET N = INT(N*100+.5)/100
210 LET N$=STR$(N)
215 LET N$=MID$(N$,2,LEN(N$)-1)
220 LET P=0:LET C$="":LET D$=""
230 FOR I=1 TO LEN(N$)
240   IF MID$(N$,I,1)="." THEN LET P=I
250 NEXT I
260 IF P=0 THEN LET P=LEN(N$)+1:LET N$=N$+".0"
270 LET N$=N$+"0"
280 LET C$=MID$(N$,P,3)
290 IF P=1 THEN LET D$="0":GOTO 430
300 LET N$=MID$(N$,1,P-1)
310 IF LEN(N$)<=3 THEN LET D$=N$: GOTO 430
320 LET T = INT(LEN(N$)/3)-1
330 LET L=LEN(N$)-(T+1)*3
340 IF L=0 THEN GOTO 360
350 LET D$=LEFT$(N$,L)+","
360 LET L=L+1
370 IF T=0 THEN GOTO 420
380 FOR I = 1 TO T
390   LET D$=D$ + MID$(N$,L,3)+","
400   LET L = L + 3
410 NEXT I
420 LET D$=D$+MID$(N$,L,3)
430 LET P$="$"+D$+C$
440 RETURN
```

Figure S.11: STR$—Sample Program, continued

The rest of the subroutine is devoted to formatting the string according to the following specifications:

1. The value will contain a decimal point, followed by exactly two digits. If the original number did not include cents, two zeros will be inserted after the decimal point.
2. Every three dollar digits (moving left from the decimal point) will be separated by commas.
3. A dollar sign will be inserted at the beginning of the string.

Here is a brief paraphrase of the subroutine, which will help you understand its various algorithms:

— ***Lines 230 to 270***. Search for a decimal point in the string. If one exists, store its position in the variable P. If there is no decimal point, add the characters ".00" to the end of the string; set P to one greater than the length of the original string.

— ***Lines 280 to 310***. Separate the dollar digits from the cents. Assign the decimal point and two decimal digits to the variable C$, and the dollar digits to N$. If N$ contains three digits or fewer, no commas are needed; in this case assign the dollar digits to D$ and jump down to the bottom of the subroutine.

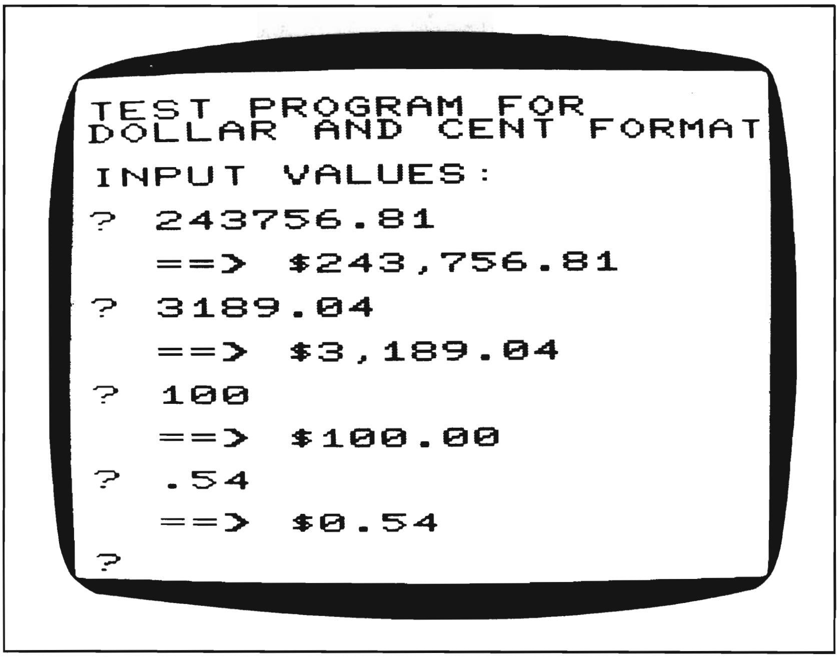
— ***Lines 320 to 420***. Determine how many commas are required for the dollar digits, and insert them correctly into the string. (A new string, complete with properly placed commas, is built from left to right and stored portion by portion in the variable D$.)

— ***Lines 430 and 440***. Combine D$ and C$ with a leading dollar sign, and assign the concatenated strings to P$. Return to the calling program.

Notice that this subroutine makes frequent use of the MID$ function to access portions of strings; for example:

```
280 LET C$ = MID$(N$,P,3)
```

This statement assigns three characters of the string N$, starting from position P in N$, to the variable C$. N$ of course remains unchanged. (*See the entry under* MID$.)



```
TEST PROGRAM FOR
DOLLAR AND CENT FORMAT
INPUT VALUES:
? 243756.81
    ==> $243,756.81
? 3189.04
    ==> $3,189.04
? 100
    ==> $100.00
? .54
    ==> $0.54
?
```

Figure S.12: STR$—Sample Output

Notes and Comments

— If a number is over nine digits long, the computer displays it in scientific notation. For example, the following PRINT statement:

```
PRINT 1000000000
```

results in the display:

```
1E+09
```

Likewise, the statement:

```
LET N$ = STR$(1000000000)
```

will assign the string value:

```
"1E+09"
```

to the variable N$. For this reason, the dollar-and- cent formatting subroutine described above will only work properly for up to nine-digit dollar values. In addition, you will begin noticing a loss of precision (rounding off of cents) when you use this program to format dollar values of eight digits or greater.

Subroutine (computer vocabulary)

A subroutine is a block of program instructions, set off from the main body of the program, and designed to be performed via the GOSUB statement. (*See the entry under* GOSUB.)

SYS (command word)

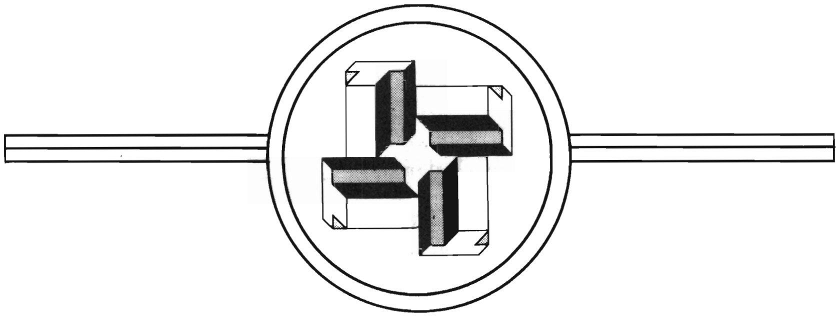
The SYS command sends control to a specified machine-language routine. The syntax of SYS is:

```
SYS M
```

where M is a decimal memory address. This command instructs the computer to perform the machine-code routine located at address M. If you write such a routine, you must include in it a machine-language RTS command ("return from subroutine") to return control to the BASIC program.

Notes and Comments

— The USR command also sends control to a machine language routine, and allows the passing of a parameter value. (*See* USR.)



TAB (function)

TAB is a programming *function* that works exclusively with the PRINT command to determine the starting column of a display element. TAB takes a single numeric *argument* in the following form:

```
PRINT TAB(N); "INFORMATION"
```

This statement instructs the computer to position the first character of the display element (in this case, the *string* "INFORMATION") at column N of the current line. The *value* N may be expressed as a *literal* numeric *value,* a numeric *variable*, or an *arithmetic expression*. The semicolon after TAB is optional.

Since the VIC-20 and the Commodore 64 computers present output screens of different widths, the effect of TAB will differ for the two computers. The argument, N, of TAB refers to a column number from 0 to L, where L is the number of the last column on the screen:

For the VIC-20, L = 21.
For the Commodore 64, L = 39.

In either case, if the argument N is greater than L, the computer tabs over an entire line for every multiple of (L + 1), and then tabs forward to the column indicated by N *modulus* (L + 1). (The *modulus* operation represents the remainder from the division of two integers.) For example, consider the statement:

```
PRINT TAB(95); "X"
```

On the VIC-20 computer, this command will cause the computer to jump over four lines of the screen—TAB(88)—and then to display the X at column 7 of the fifth line (95 *modulus* 22 = 7). On the Commodore 64 computer, this command will cause the computer to jump over two complete lines of the screen—TAB(80)—and then to display the X at column 15 of the third line (95 *modulus* 40 = 15).

The legal range for the argument, N, of TAB is:

$$0 <= N <= 255$$

Sample Program

The short program in Figure T.1 allows you to experiment with different arguments of TAB on either the VIC-20 or the Commodore 64 computer. When you run the program, the question:

```
TAB TO?
```

will appear at the top of the screen. If you enter any number from 0 to 255, the screen will clear, and an asterisk will appear at the tab position corresponding to the number you typed. Next to the asterisk will be a TAB expression indicating the position; for example:

```
* TAB( 95 )
```

The program forms a repeating loop, allowing you to examine as many TABs as you wish.

Line 30 of the program reads the input value into the variable T. Line 50 then tabs forward to T and prints the message:

```
50 PRINT TAB(T); "* TAB(";T;")"
```

```
10 REM ** TAB DEMO
20 PRINT CHR$(147)
30 INPUT "TAB TO";T
35 IF T>255 GOTO 30
40 PRINT CHR$(147)
50 PRINT TAB(T);"* TAB(";T;")"
60 PRINT : PRINT
70 INPUT "CONTINUE";A$
80 GOTO 20
```

Figure T.1: TAB—Sample Program

TAN (function)

Given an angle expressed in radians, the TAN *function* returns the tangent of the angle. The tangent function approaches infinity as its *argument* approaches odd multiples of ± π/2.

Sample Program

The program shown in Figure T.2 displays a series of tangent values for arguments ranging from -1.6π to + 1.6π. The control *variable* I of the FOR loop at line 70 determines the arguments of TAN in line 80. The output from the program appears in Figure T.3.

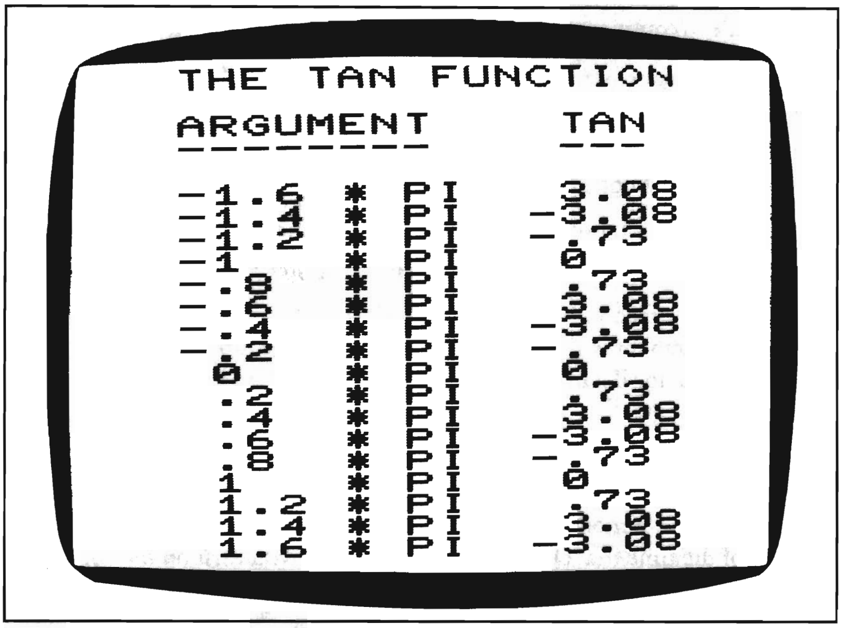
Notes and Comments

— Figure T.4 shows a crude graph of the tangent function from $x=0$ to $x=2\pi$, produced on the VIC-20 computer.

— *See the entries under* SIN and COS for more information about the trigonometric functions.

```
  5 DEF FNR(X)= INT(100*X+.5)/100
 10 PRINT CHR$(147)
 20 PRINT TAB(3); "THE TAN FUNCTION"
 30 PRINT
 35 PRINT "   ";
 40 PRINT "ARGUMENT    TAN"
 45 PRINT "   ";
 50 PRINT "--------    ---"
 60 PRINT
 70 FOR I = -1.6 TO 1.6 STEP .2
 73 LET I = FNR(I)
 75 PRINT "   ";
 80 PRINT I;TAB(8);"* PI";TAB(14); FN R(TAN(I*3.1416))
 90 NEXT I
100 GET A$
110 IF A$="" GOTO 100
```

Figure T.2: TAN—Sample Program



```
THE TAN FUNCTION
ARGUMENT          TAN
---------         ---

-1.6  *  PI       3.08
-1.4  *  PI      -3.08
-1.2  *  PI      -.73
-1    *  PI       0
-.8   *  PI       .73
-.6   *  PI       3.08
-.4   *  PI      -3.08
-.2   *  PI      -.73
 0    *  PI       0
 .2   *  PI       .73
 .4   *  PI       3.08
 .6   *  PI      -3.08
 .8   *  PI      -.73
 1    *  PI       0
 1.2  *  PI       .73
 1.4  *  PI       3.08
 1.6  *  PI      -3.08
```

Figure T.3: TAN—Sample Output

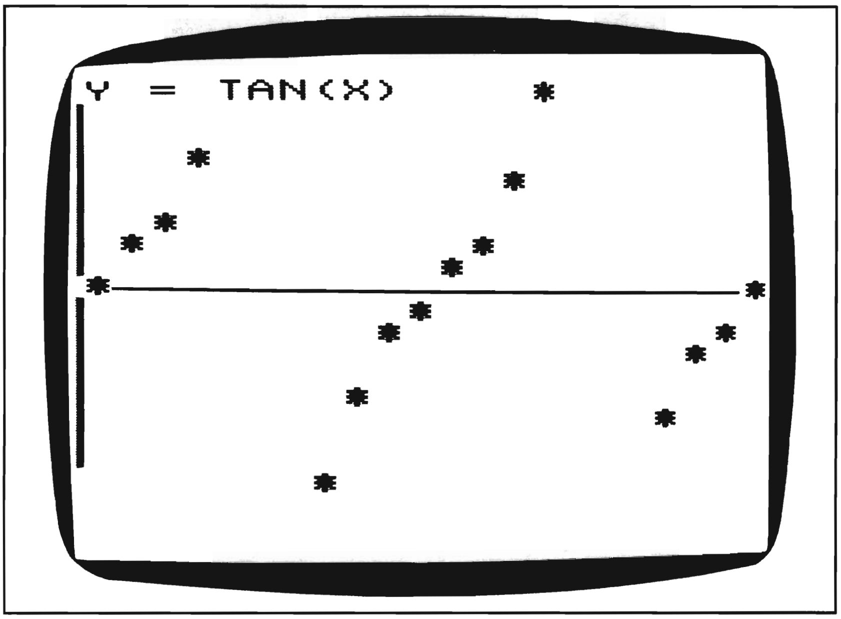


Figure T.4: TAN—Plotted Graph

THEN (command word)

Every IF statement must have a THEN clause. If the logical expression after the word IF is evaluated as true, then the computer will perform the action expressed after THEN. A BASIC command always follows THEN:

```
IF (logical expression) THEN (command word)
```

If THEN is followed by more than one statement, in the following form:

```
IF (logical expression) THEN (statement-1) : (statement-2)
```

then both statements will be performed if the logical expression is true, and neither statement will be performed if the logical expression is false. *See the entry under* IF for details.

TI (time function)

Both the Commodore 64 and the VIC-20 have a built-in clock that keeps track of the time that elapses from the moment you turn on the computer. The time is kept in "jiffies," units of 1/60th of a second. The TI *function* reads the clock and returns its current value. Thus, you can use statements such as the following to find out how long your computer has been operating:

Number of jiffies since start-up:

```
PRINT TI
```

Number of seconds since start-up:

```
PRINT TI/60
```

Number of minutes since start-up:

```
PRINT TI/3600
```

Number of hours since start-up:

```
PRINT TI/216000
```

A word of caution: you cannot rely on the accuracy of the built-in clock after you have used the cassette recorder for saving or loading programs or data.

Sample Program

The program shown in Figure T.5 illustrates an important use of the TI function: creating pauses of a specified duration during a program run. The subroutine at line 100 creates the pause; the "main program" section, from lines 5 to 60, is designed simply to test the pause subroutine. Line 30 reads, from the keyboard, the number of seconds that you want the subroutine to measure:

```
PAUSE TEST

HOW MANY SECONDS?
```

As soon as you enter a number, the program will pause for the number of seconds indicated. At the end of the pause the program displays the message:

```
DONE.
```

on the screen. You can test the program for several different pause lengths; if you have a clock that measures seconds, you will see that the program is quite accurate.

The main program reads the number you enter from the keyboard into the variable S, for seconds, and then immediately calls the subroutine at line 100. The pause subroutine is very simple in design. It begins by storing the current value of TI in the "holding" variable, H:

```
100 LET H = TI
```

It then calculates the length, in jiffies, of the desired pause:

```
110 LET J = S * 60
```

You can see that H + J will be the value of TI at the end of the pause. Thus, all the subroutine needs to do is continually test the value of TI until it

```
  5 PRINT CHR$(147)
 10 PRINT "PAUSE TEST"
 20 PRINT
 30 INPUT "HOW MANY SECONDS";S
 40 GOSUB 100
 50 PRINT "DONE."
 60 GOTO 20
100 LET H=TI
110 LET J=S*60
120 IF TI>=H+J THEN RETURN
130 GOTO 120
```

Figure T.5: TI—Sample Program

reaches H + J; at this point control is returned to the main program:

```
120 IF TI > = H + J THEN RETURN
130 GOTO 120
```

Line 130 forms a loop that ends only when the pause is over.

TI$ (time function)

TI$ is a "real clock" *function*; it supplies a six-digit *string* value representing a 24-hour clock. The six digits of the string indicate hours (HH), minutes (MM), and seconds (SS), as follows:

```
HHMMSS
```

Thus, for example, the TI$ value 112415 means 11:24:15 a.m. The TI$ value 164830 means 4:48:30 p.m.

To use the TI$ function as a time-of-day clock, you must first set it to the current time. You can do this in an assignment statement, just as though TI$ were a simple string variable. For example, let's say the current time is 12:30 in the afternoon; you can set the TI$ function as follows:

```
LET TI$ = "123000"
```

After you have set TI$, you can examine it whenever you want to find out the time-of-day:

```
PRINT TI$
```

Like the TI function, TI$ loses reliability after cassette tape recorder operations, such as saving or loading programs.

Sample Program

The program shown in Figure T.6 will turn your computer into a digital clock, displaying the time on the screen. When you run the program, it begins by asking you to input a six-digit time string to initialize TI$:

```
CLOCK PROGRAM
TIME STRING?
```

You enter the current time, and the clock display begins. For convenient

```
 5 PRINT CHR$(147)
10 PRINT "CLOCK PROGRAM"
20 INPUT "TIME STRING"; TI$
30 PRINT CHR$(147)
35 FOR I = 1 TO 10: PRINT: NEXT I
40 PRINT TAB(7); LEFT$(TI$,2);":";
50 PRINT MID$(TI$,3,2);":";
60 PRINT RIGHT$(TI$,2);
70 PRINT CHR$(19)
80 GOTO 35
```

Figure T.6: TI$—Sample Program

reading, the program displays the hours, minutes, and seconds separated by colons; for example:

```
12:15:38
```

The program works by continually accessing the value of TI$ and displaying that value at the same location on the screen. Line 20 reads the keyboard input directly into TI$:

```
20 INPUT "TIME STRING"; TI$
```

Line 35 skips down 10 lines; and lines 40, 50, and 60 print the hours, minutes, and seconds, respectively, using the LEFT$, MID$, and RIGHT$ functions to access portions of the TI$ string:

```
40 PRINT TAB(7); LEFT$(TI$,2); ":";
50 PRINT MID$(TI$,3,2); ":";
60 PRINT RIGHT$(TI$,2)
```

Line 70 repositions the cursor at the upper-left corner of the screen, and line 80 forms a loop out of the whole display process:

```
70 PRINT CHR$(19)
80 GOTO 35
```

The time is displayed in the middle of the VIC-20 screen. If you use this program on the Commodore 64, you might want to change the TAB function in line 40 to TAB(16).

TO (command word)

TO is part of the syntax of a FOR loop. Specifically, TO indicates the range of the control *variable*; for example:

```
FOR I = 1 TO 30
```

In the performance of this FOR loop, the variable I will be incremented in value from 1 to 30. *See the entries under* FOR and STEP for further information.

User-Friendly (computer vocabulary)

A program that helps, rather than confuses, the person who is using it is described as "user-friendly." Some of the elements of a user-friendly program are:

— a clear description of the options available to the user and the methods of choosing those options;

— precise and clear input prompts, telling the user what kind of data is expected from the keyboard and when;

— efficient, simple ways of recovering from input errors.

For further discussion, *see the entries under* GOSUB, GET, and LEFT$.

USR (function)

The USR *function* allows you to call a *machine code subroutine* from within a BASIC program. Unlike the SYS command, USR lets you send a real numeric value to the subroutine and get a value back from the subroutine. The value you send is the *argument* of USR, and the value you get back is the value of USR on return from the subroutine.

The format of USR is:

USR(V)

where V is a *literal* numeric *value*, a *variable*, or an *arithmetic expression* that evaluates to a real number. USR always stores the value V in a fixed memory location, so the value is available for use in the machine-language routine called by USR. Since USR is a function, it cannot, of course, stand alone, but must be part of a program statement. For example, USR might appear as part of a PRINT statement:

```
PRINT USR(V)
```

or an assignment statement:

```
LET T = USR(V)
```

Specifically, the action of USR is as follows:

1. It stores the real value V in memory locations 97 to 102, the "floating-point accumulator."
2. It then performs a machine language JSR ("jump to a subroutine") to a specific memory location, which in turn contains a "vector" that directs control to the machine-language routine that is to be performed.

The memory location of this USR "vector" depends on which of the Commodore computers you are using. On the VIC-20, it is memory address 0; on the Commodore 64, it is address 784. These addresses contain a machine language JMP ("jump") command; before you invoke the USR function, you must POKE the starting address of your machine-language routine into the two locations following the JMP command: locations 1 and 2 for the VIC-20; locations 785 and 786 for the Commodore 64.

For example, let's say you have written a machine-language routine for the VIC-20, and stored it in memory locations starting at decimal address 7424. The hexadecimal equivalent of this address is 1D00. You must POKE the lower two digits of the hexadecimal address into memory location 1 and the high digits into memory location 2; the following two commands would do the job:

```
POKE 1, 0
POKE 2, 29
```

Notice that the decimal equivalent of 1D, the high digits of the hexadecimal address, is 29. Thus, the computer will read the values stored in locations 1 and 2 as address 1D00.

On the Commodore 64, you could POKE the same starting address into locations 785 and 786:

```
POKE 785, 0
POKE 786, 29
```

To test these vector values, POKE a simple RTS command (op code 96, decimal) into memory location 7424:

```
POKE 7424, 96
```

Now you can enter any USR command you want; the value returned by USR will simply be the argument of the function itself. For example, the command:

```
PRINT USR(25)
```

will result in a display of the value 25 on the screen. This is because the value returned by USR is taken from the "floating-point accumulator," the same locations where the argument of USR is stored when the function is invoked. Since your machine-language routine in this case (simply RTS) does nothing to change this value, it is returned intact by the function.

Notes and Comments

— The SYS command sends control directly to a machine language routine at a specified address. (*See the entry under* SYS.)

— The NEW command changes neither a machine-language routine you have stored in memory, nor the "vector" address that points to that routine.

VAL (function)

The VAL *function* supplies the numeric equivalent of a *string*. VAL takes the form:

```
VAL(S$)
```

where S$ is a string that can be converted into a number. S$ may consist of digits and any of the following:

- — a decimal point;
- — a leading plus sign or minus sign;
- — a *scientific notation* format, with the letter E.

For example, any of the following strings would be valid *arguments* for VAL:

```
"1235"
"9862.89"
"+7.3"
"-81"
"3.5E+12"
"1E-15"
```

If VAL receives a string argument whose first character cannot be converted into a numeric value, the function returns value of zero. For example:

```
VAL("J1234")
```

would result in zero. If some character other than the first in the argument is nonnumeric, VAL will return the numeric equivalent of all the characters up to the first nonnumeric character. For example:

VAL("121.3A123")

would result in the value 121.3.

Sample Program

The program shown in Figure V.1 is a test program for a numeric input validation routine. The input *subroutine* starts at line 100. This routine attempts to avoid input errors by *not accepting* invalid characters—or invalid combinations of characters—typed from the keyboard. You can see from the dense web of IF statements in this subroutine that it is a complex task.

```
 10 REM ** INPUT TEST
 15 PRINT CHR$(147)
 20 PRINT "TYPE ANY NUMBER."
 30 PRINT "NONNUMERIC CHARACTERS"
 40 PRINT "WILL NOT REGISTER.
 50 PRINT
 60 PRINT ": ";
 70 GOSUB 100
 75 PRINT
 80 PRINT " "; N
 90 GOTO 50
100 REM ** INPUT
110 LET F=0 : LET T=NOT F
120 LET S=T : LET P=F : LET E=F
130 LET N$=""
140 GET C$
150 IF C$="" GOTO 140
155 IF C$=CHR$(20) THEN GOSUB 300 : GOTO 140
160 IF C$>="0" AND C$<="9" GOTO 210
170 IF (C$="+" OR C$="-") AND (S=F FOR N$="") THEN LET S=T :
    GOTO 210
180 IF C$="." AND P=F THEN LET P=T : GOTO 210
190 IF C$="E" AND E=F AND N$<>"" THEN LET E=T : LET S=F :
    LET P=T : GOTO 210
```

Figure V.1: VAL—Sample Program

```
195 IF C$=CHR$(13) THEN GOTO 230
200 GOTO 140
210 PRINT C$;
215 LET N$=N$+C$
220 GOTO 140
230 LET N=VAL(N$)
240 RETURN
300 REM ** DELETE KEY
310 IF LEN(N$)=0 THEN RETURN
315 LET H$=MID$(N$,LEN(N$),1)
320 PRINT CHR$(20);
330 LET N$=MID$(N$,1,LEN(N$)-1)
340 IF H$="E" THEN LET E=F : LET S=T : LET P= F
350 IF H$="." THEN LET P=F
360 IF H$="-" OR H$="+" THEN LET S=F
370 RETURN
```

Figure V.1: VAL—Sample Program, continued

The "main program" section of the program allows you to test the subroutine. When you run the program, the following message will appear at the top of the screen:

```
TYPE ANY NUMBER.
NONNUMERIC CHARACTERS
WILL NOT REGISTER.
```

You will also see a colon (:) as an input prompt, and the program will wait for you to begin typing a number. Any valid numeric character (including "+", "-", ".", and "E", if they are entered at appropriate positions in the input) will be "echoed" on the screen and included as part of the input value. Any *nonnumeric* character that you type will not appear on the screen—the program rejects such characters as invalid. In addition, the program imposes the following restrictions on your input:

1. You may enter the letter E once in the input, to indicate scientific notation; but E may not be the first character of the input.
2. You may enter one decimal point as part of the input. A decimal point is not allowed after E.
3. You may enter a plus or minus sign as the first character of the input and/or as the first character after the E.
4. You may use the DEL key to delete any character that you have

typed, but you may not delete further backward than the first character of your input.

5. Pressing the RETURN key completes your input.

If you attempt to enter any invalid characters—i.e., characters that do not follow these specifications for correct numeric input—the program's reaction is perhaps the most elegant reaction of all: it does nothing. Your invalid character will not be "echoed" on the screen, and, more importantly, will not be stored as part of the input. After you press RETURN, your input number is displayed again on the screen for you. The program allows you to test this input routine with as many input values as you want to try.

The input routine's basic technique is simple; it involves the GET and VAL functions and the string concatenation operation.

The routine GETs each character from the keyboard:

```
140 GET C$
150 IF C$ = "" GOTO 140
```

and then subjects the character to a series of validation tests to see if it can be accepted as part of the numeric input. If any one test fails—showing that the character is invalid—control returns to line 140, and the program simply waits for the next input character. However, if the character C$ is a valid one, the program "echoes" the character on the screen:

```
210 PRINT C$;
```

and adds the character to the end of the input string N$:

```
220 LET N$ = N$ + C$
```

Then, once again, control returns to the top of the routine for another input character:

```
220 GOTO 140
```

When you finally press the RETURN key, the program uses the VAL function to convert the input string N$ into a numeric value, N:

```
230 LET N = VAL(N$)
```

and returns control to the calling program:

```
240 RETURN
```

The numeric variable N, then, contains the input value.

In the process of validating the input characters, this subroutine makes use of several "Boolean" variables to keep track of the current status of the input string. Some programming languages actually have variables of type Boolean; these are variables that take one of two values: true or false.

BASIC has no such variable type, but you can create variables that will serve the same function. Since BASIC evaluates logical expressions to a numeric value (0 for false; -1 for true—*see the entry under* IF for details), you can reserve certain variables that you know will always contain one of these two values. Then, when you need to, you can test the current values of the variables and make decisions based on those values.

This subroutine can accept certain input characters only at certain points in the input validation process. For example, the character "E" may be accepted only once in the input, but not as the first character. Other characters that may be accepted only at restricted points in the input are "+", "-" and ".". To keep track of whether or not one of these characters can be accepted at any given point, the subroutine uses the three "Boolean" variables E (for exponent), S (for sign), and P (for decimal point). In addition, the program sets up the two variables T and F, for true and false:

```
110 LET F=0 : LET T=NOT F
```

The variables E, S, and P always contain either the value of F or the value of T. The value of F in one of these variables means that the opportunity to use the corresponding character in the input string ("E", ".", "+", or "-") has not yet been taken, and the character is therefore valid if it is entered from the keyboard. The value of T, on the other hand, means that the corresponding character is invalid, and should not be accepted from the keyboard. The use of such "Boolean" variables can often be very helpful in programs that involve complex logical decisions.

In summary, then, this program reads numeric input from the keyboard character by character, validating each character as it is entered, and building the input string N$ from all the valid characters. When the RETURN key is pressed, the program uses VAL to supply the numeric equivalent of N$, and stores the numeric input value in N. The obvious question about all this is, why go to the trouble of writing such a long and complicated input routine when a simple INPUT statement does essentially the same job? After all, we could simply write the statement:

```
100 INPUT N
```

to read any numeric input value from the keyboard.

Obviously, an input routine like the one shown in this program would not be necessary in most programming situations. The INPUT command is usually perfectly adequate. Sometimes, however, you may encounter programming tasks where the amount or the significance of numeric input requires that you take greater control over input validation than the INPUT command allows. In such situations you may want to write a routine such as this one, relying on GET and VAL rather than INPUT. In addition to generally increased control, this routine gives you the following advantages:

— You can use it to avoid the *error message* the INPUT statement displays on the screen in the event of invalid numeric input. Recall that INPUT rejects invalid numeric values by displaying the following message:

```
?REDO FROM START
?
```

If you have planned a carefully designed input screen, where each line and space on the screen is significant, you will want to avoid at all costs the intrusion of this error message.

— You can supply any input prompt that you think appropriate. Recall that INPUT always displays a question mark on the screen as the prompt. Using another input routine, you can change that prompt to a colon, as in this example, or to any other character or set of characters that you might want to use.

Variable (computer vocabulary)

Think of a variable as a place set aside in the computer's active memory for a data element of a specified type; in your program the variable is represented by a given name. The variable name itself indicates the type of data the variable can store; the last character of the variable name is the type indicator:

— $ indicates a string variable;

— % indicates an integer variable;

— a letter or a digit indicates a real-number variable.

All variable names must begin with a letter from A to Z. A variable name may be of virtually unlimited length, but only the first two characters (plus the type indicator) are significant. The second character may be a letter or a digit; the computer ignores characters after the second, except for the last character, if it is a type indicator. For this reason, BASIC will treat the following pairs of variables as identical:

AM and AMOUNT
AN$ and ANSWER$
QU% and QUANTITY%

The name of a BASIC command word or function name may not be used as or embedded in a variable name. Such a variable name will cause a syntax error. For example, the variable name ERROR would be invalid because it contains the BASIC word OR.

VERIFY (program storage command)

The VERIFY command compares the current program in the computer's memory with a specified program file on cassette or disk. If the two programs are identical, the computer displays the message:

```
OK
```

If there is any difference between the two programs, however, you will see the message:

```
? VERIFY
  ERROR
```

It is a good idea to use the VERIFY command each time you store a program on cassette or disk to make sure that the storage operation was successful and error-free.

The format of the VERIFY command is as follows:

```
VERIFY "FILE NAME", D
```

where the file named within quotation marks is the one you want to compare with the current program in memory. The device number, D, is 1 for a cassette, and 8 for a disk. If the device parameter is omitted, the "default" device number is 1.

WAIT (command word)

WAIT is a command whose purpose is to create a pause in the action of a BASIC program. The command's design makes it an uncharacteristically obscure BASIC instruction; it is only useful in rather special circumstances involving I/O operations.

The syntax of WAIT is:

```
WAIT M, N1, N2
```

where M is the address of a memory location; and N1 and N2 are integer values from 0 to 255. (N2 is an optional parameter, and is set to zero if it is not present in the statement.) WAIT deals with the binary equivalents of N1 and N2, and the binary value stored in M. Technically, it performs an XOR ("exclusive OR") on the two values, N2 and the value stored at address M; then a logical AND on N1 and the result of the XOR operation. WAIT creates a pause in the program performance until the binary result of these two operations contains at least one nonzero bit.

Notes and Comments

— If you just want to make your program pause at a certain point in the action, there are two ways to do so on the Commodore computers. The more accurate technique, for occasions when you want a timed pause, is to use the TI function; *see the entry under* TI for a complete description of this technique. A simpler technique is to perform an empty FOR loop:

```
100 FOR I = 1 TO N
110 NEXT I
```

You can experiment with different values of N, in line 100, to produce pauses of different lengths. If you set N to 1000, the pause will probably last for about two or three seconds. The sample program under the heading STEP shows an example of this kind of pause loop.

INDEX

This index provides a cross-referencing tool for each word in the BASIC vocabulary. You will, of course, find the most complete coverage of any given word under the word's own entry (page numbers shown here in **boldface** type); however, in many cases you will discover additional insights and examples under other entries. The purpose of this index, then, is to help you locate additional information about any word that you may want to study in detail.

THE COMMODORE 64/VIC-20 BASIC HANDBOOK

This handy computer-side reference will make programming your Commodore 64 or VIC-20 easier, whether you're an experienced programmer or a first-time user.

This unique book lists and explains, in alphabetical order, each one of the Commodore computer's BASIC commands and functions.

Explanations include special tips and suggestions for using the BASIC vocabulary to make programming as simple and efficient as possible, *information you won't find in other manuals.*

Sample programs show you how to use the commands in their proper syntax, and sample output screens show exactly what the commands do.

Learn the best way to use:

- FOR/THEN Loops
- IF/THEN Statements
- Subroutines

Use special mathematical functions such as:

- ABS for calculating absolute value
- EXP for calculating exponents

You'll find clear and simple explanations for important general vocabulary terms such as:

- Algorithm
- Array
- Interactive

Find out what an Assignment Statement is and how you can make one in BASIC.

This book makes it easy to program your Commodore 64 or VIC-20 to do a multitude of convenient home and office tasks.

ABOUT THE AUTHOR:

Douglas Hergert is a freelance writer. He is the author of these SYBEX publications: *The Apple II BASIC Handbook, The Timex/Sinclair 1000 BASIC Handbook, Your Timex/Sinclair 1000 and ZX81, BASIC for Business,* and *Mastering VisiCalc.* He is the coauthor of *Apple Pascal Games* and *Doing Business with Pascal,* and the translator of X.T. Bui's *Executive Planning with BASIC.*

ISBN 0-89588-116-0